Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DE COMPUTADORES

Clobber: Arquitectura para streaming de aplicaciones nativas sobre HTTP

Estudante: Xabier Iglesias Pérez
Dirección: Diego Darriba López

A Coruña, xuño de 2021.

Para RO, por hacer que valga la pena.

Agradecimientos

A mis abuelos por cuidarme y quererme, a mis padres por hacerme libre y a Charo por mi vida entera. A Diego por ser un tutor increible. A la ASOC y a mis compañeros de carrera. En especial a Alonso, Natalia, Iñaki, Martín y Javi. Siempre estaréis conmigo.

Sorry!

Clobber: *1. Propiedad de un registro que puede cambiar sin previo aviso durante la ejecución de una subrutina ensamblador. 2. Producir daño financiero.*

Resumen

Este proyecto fin de grado plantea una solución sencilla, dinámica y eficiente a la necesidad de actualización del hardware para equipos de usuario a intervalos regulares, empleando para ello software de virtualización y una arquitectura de microservicios. La infraestructura desplegada a modo de demostración permite, empleando el navegador como interfaz, interactuar con un sistema operativo en red de manera visual, rápida y segura, contando con la capacidad teórica del equipo anfitrión. El objetivo consiste en delegar en el cliente únicamente la necesidad de renderizar la parte visual. En este documento definiremos arquitectura, herramientas y software que integran el producto final; apoyándonos para ello en los conocimientos adquiridos durante el transcurso del grado. El código resultante de este proyecto se encuentra disponible bajo licencia GPLv3 en https://github.com/TretornESP/Clobber/.

Abstract

This dissertation proposes a lightweight and optimized solution for the constant need to upgrade consumer hardware due to planned obsolescence. In order to achieve this goal we take advantage of modern virtualization software to develop a service that enables engineers to endow their projects with the resources of a fully featured operating system and theoretically unlimited computing power. The goal is to limit consumer hardware to mere input/output devices while processing and storage tasks are shifted to centralised servers where horizontal scalability would be possible and a more responsible disposal protocol be performed. This project addresses the architecture and software part of the service. The code of this project is available under GPLv3 licensing in https://github.com/TretornESP/Clobber/.

Palabras clave:

- Docker
- Obsolescencia programada
- API
- Linux
- Virtualización
- Arquitectura de microservicios
- Escalabilidad
- Tolerancia a fallos
- Redes de almacenamiento

Keywords:

- Docker
- Planned obsolescence
- API
- Linux
- Virtualization
- Microservice architecture
- Scaling
- Fault tolerance
- Storage networks

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este primer capítulo introduciremos el proyecto a través de la motivación que lo impulsa, posteriormente enunciaremos la solución propuesta para finalizar enmarcándola como parte de un proyecto mayor.

1.1 Motivación

Cuando *Isaac Asimov* escribió el *Ciclo de la Tierra*, primer libro de *Serie de la Fundación*, era normal que situase sus novelas en un futuro donde las máquinas cobraban rasgos de humanidad, destacando de entre todos ellos el temor hacia la propia muerte. *Asimov expone así que la evolución parte desde lo inerte y automático hacia lo emotivo e irracional.*

Más de setenta años después, nuestra sociedad ha ido avanzando en sentido opuesto, tratando por cualquier medio de ocultar su propia condición mortal en un retorno a lo inerte a través de la lógica. El consumismo, la polarización política y las guerras por territorios incluso carentes de recursos cursan como alternativa a la religión, que palidece ante el inexorable avance de la ciencia.

La evolución tecnológica sobrepasa en celeridad a la natural. Esta diferencia es tan elevada que podemos considerar la variación de las capacidades físicas humanas como invariables en el tiempo.

Considerando la evolución del hardware que interactúa con nuestros sentidos, podríamos vislumbrar un futuro donde la capacidad de manufacturación alcanzase la excelencia abso-

luta. Un ejemplo donde este fenómeno ya se ha producido es el audio digital con respecto al rango de frecuencias audibles: el teorema de muestreo de *Nyquist-Shannon* [1] afirma que podremos reconstruir una señal analógica *muestreada* al doble de su frecuencia máxima sin perder resolución. Teniendo en cuenta que un humano rara vez es capaz de percibir frecuencias superiores a los *20 kHz*, podemos afirmar que hemos alcanzado un límite físico en el audio de *40 kHz* [2].

Este fenómeno no se produce en el software: no parece que las necesidades de cómputo vayan a dejar de crecer a corto plazo. Nuestro objetivo es dividir estos dos mundos desplazando la carga de proceso desde los dispositivos de entrada/salida hacia servidores remotos y centralizados, donde podamos optimizar mucho mejor el uso de los recursos disponibles, implementar una arquitectura escalable y aplicar un plan de reposición responsable para el hardware antiguo.

El fin de este trabajo es atacar un pilar de la sociedad actual tal como es la obsolescencia programada, por medio de una solución tecnológica *sencilla, duradera y retrocompatible* con parte del hardware que había sido desechado.

La tecnología puede salvarnos de todo menos de nosotros mismos.

1.2 Solución propuesta

Como concepto global, Clobber se compone de tres pilares:

1. Un dispositivo *hardware* para el consumidor final especializado en funciones de entrada/salida.

2. Un sistema operativo optimizado, minimalista y con baja demanda de recursos, centrado en la conectividad inalámbrica.

3. Un servicio distribuido que otorgue la capacidad de proceso al dispositivo anterior.

El primer pilar acabaría con la diferenciación entre teléfonos, tablets y PCs. Éstos pasarían a ser diferentes formas de interactuar con un mismo entorno y sua labor consistiría en manejar sensores (acelerómetro, GPS, micrófono, etc), enviar y recibir paquetes de red y mostrar datos por pantalla.

El segundo pilar consiste en un kernel específico: partir de un modelo diferente al clásico *UNIX* nos permitiría centrar esfuerzos en módulos como el stack de red mientras aliviamos carga de proceso obviando elementos como el sistema de ficheros y permisos de usuario. Rescataríamos de *Linux* los conceptos de *kernel* monolítico (ya que intentamos parecernos lo máximo posible a un firmware) y el sistema IOCTL por su extensa aceptación entre las empresas que programan device drivers.

El último elemento engloba toda la parte de la infraestructura de servidores y máquinas virtuales. Será introducido en capítulos posteriores al ser este **el objetivo de nuestro trabajo**.

1.3 Alcance del proyecto y objetivos

El producto que nace a partir de este documento consiste en una arquitectura compuesta por servicios estructurados en tres niveles diferenciados: web, virtualización y almacenamiento.

- El **nivel web** es la abstracción que contiene las páginas de administración, portales de autenticación y documentación, al ser accesible líbremente desde internet decidimos aislarlo de cara a la futura inclusión de un cortafuegos que filtre su comunicación con niveles inferiores.
- El **nivel de virtualización** es el *músculo* de Clobber ya que aloja cada una de las instancias Clobber. Se ha aislado para facilitar la elasticidad de la solución. Podemos asimilar la carga de trabajo total de la infraestructura a la suma de las cargas de trabajo de todos los servidores presentes en este nivel. En el convivirán varios contenedores Docker. Cada uno de estos contenedores contará con software específico para acceder a funciones de red, almacenamiento remoto mediante iSCSI y labores de administración. En adelante nos referiremos a estos contenedores como instancias Clobber.
- El **nivel de almacenamiento** contiene los datos de todos los usuarios, en el reside el estado de cada máquina, repartido entre varios servidores. Existen multitud de ventajas al aislarlo del resto de servicios, entre ellas se encuentran:
  - Mayor escalabilidad.
  - Seguridad incrementada.
  - Copias de seguridad transparentes.
  - Facilidad para la movilidad geográfica.
  - Optimización de los equipos.

En este documento describimos también la máquina que cumple la función de *enrutador*, servidor DHCP y servidor de caché DNS.

La interacción entre los diferentes niveles queda plasmada en la Figura 1.1

Una vez acotado el proyecto, pasamos a enumerar los **objetivos específicos** que se deben cumplir para hacer operativo al producto:

1. Soportar un API capaz de describir la interacción entre el sistema operativo y la aplicación.

2. Diseñar e implementar un parser capaz de generar, a partir de un fichero, aplicaciones que interactúen con Clobber.
3. Diseñar e implementar un gestor designado para labores de *autenticación, manejo de servidores y peticiones HTTP*.
4. Elaborar un escritorio virtual a modo de demostración.
5. Programar un módulo de comandos capaz de ejecutar órdenes arbitrarias y devolver su resultado en un formato estandarizado.
6. Realizar un plan de despliegue para la infraestructura.

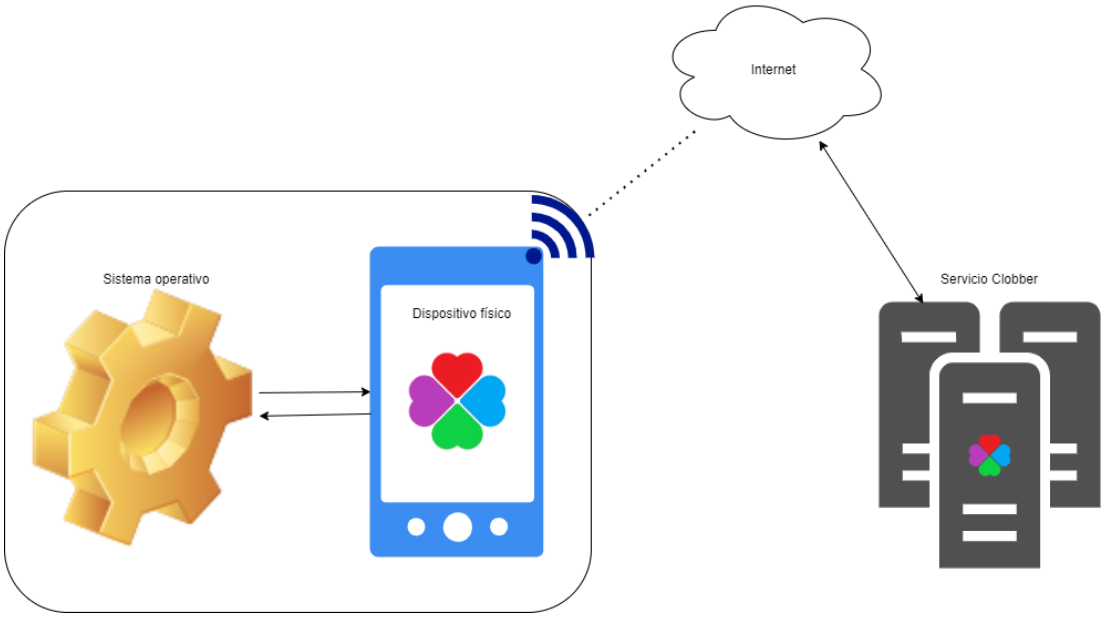

Figura 1.1: Elementos que integran la solución completa

Capítulo 2

Fundamentos y conceptos principales

Comenzamos este capítulo realizando un estudio del estado del arte. El concepto que trataremos no dista mucho del clásico esquema formado por terminales que se conectan a un mainframe. La diferencia consiste en que pasamos de tener múltiples usuarios y seguridad basada en permisos a contar con máquinas virtuales de uso individual desplazando así el nivel de aislamiento.

2.1 Descripción de conceptos clave

En esta sección hablaremos de los conceptos más importantes para nuestro proyecto.

- **Virtualización**: Mediante esta tecnología logramos dotar a cada usuario de un equipo propio sin ninguno de los costes asociados a las máquinas físicas, que son emuladas mediante software, estos equipos etéreos se denominan Máquinas virtuales. La virtualización de sistemas operativos aprovecha al máximo las posibilidades de la memoria virtual para aislar los sistemas hospedados en la máquina física pudiendo contar cada sistema con su propia tabla de páginas. La virtualización requiere de un manejo inteligente de los recursos compartidos; podemos asimilar la máquina virtual a un hilo de ejecución o a un proceso, surge pues, la necesidad de administrar este acceso a recursos mediante algoritmos para el manejo de la concurrencia.
- **Docker**: Dentro de las tecnologías de virtualización disponibles (*VMWare, VirtualBox, HyperV, etc*), hemos seleccionado Docker por su *velocidad, flexibilidad y posibilidades de automatización*. Su acercamiento a la virtualización parte de un programa que se ejecuta en la máquina anfitrión [3] al que:
  - Se le han redirigido mediante una tubería los descriptores *0, 1 y 2*.

  - Se le ha indicado que se ejecute con un identificador de proceso PID independiente.
  - Se le ha concedido un nuevo espacio de nombres UNIX Time-Sharing.
  - Se le ha asociado un nuevo espacio de nombres.
  - Se le ha asociado un nuevo CGroup.

- **API y endpoint**: Un API (*Application Programming Interface*) es un conjunto de funciones agrupadas de manera cohesiva y puestas a disposición de uno o más consumidores, normalmente programas informáticos. La manera más común de invocar a estas funciones es mediante una petición HTTP sobre una URL. Cada una de las direcciones que actúan como iniciador para una función son los denominados endpoints.

- **Flask**: Es un framework que permite desplegar *endpoints* de manera sencilla mediante la asociación de funciones Python a *urls* virtuales. Incluye soporte para las peticiones HTTP más comunes como *GET*, *POST*, *PUT*, *PATCH* y *DELETE*, manejo de autenticación y renderizado de templates entre otros.

- **AJAX y Callback**: AJAX son las siglas de *Asynchronous JavaScript and XML*. Este tipo de peticiones nos permiten solicitar datos a un servicio de manera asíncrona, es decir, no es necesario esperar a que la llamada vuelva para continuar. A cada llamada asíncrona se le asigna una función encargada ejecutarse cuando los datos estén disponibles, denominada callback.

- **SCSI**: *(Small Computer System Interface)* es una familia de protocolos de comunicación entre dispositivos de uso extendido desde mediados de los *años 80* [4]. Destaca su uso para comunicar sistemas de almacenamiento. En la actualidad el protocolo SCSI se encapsula en paquetes IP y se emplea para comunicar hardware en red. A esta implementación se la denomina iSCSI.

2.2 Alternativas a Clobber: el estado del Arte

En esta sección hablaremos de algunos esquemas alternativos que también buscan centralizar los recursos de cómputo, fabricantes que los implementan y sus ventajas e inconvenientes.

2.2.1 Múltiples usuarios locales

En el caso de que a un equipo físico se acceda de manera simultanea por parte de varios operadores estaremos ante un esquema de tipo Multiseating (Figura 2.1), es decir: aceptar la

entrada de varios teclados y ratones a la vez asociándolos al usuario deseado. Herramientas como ASTER en *Windows* o Xhepyr en *Linux* habilitan esta funcionalidad.

Los principales problemas de este esquema son:

1. **Limitaciones de hardware**: no podremos aplicar esta propuesta si el elemento de proceso está físicamente unido a un módulo de entrada/salida como es habitual en los dispositivos móviles.

2. **Limitaciones espaciales**: Los usuarios normalmente han de estar trabajando a una corta distancia del equipo y entre ellos al verse limitados por la longitud de los cables presentes en sus dispositivos de entrada.

3. **Tolerancia a fallos y escalabilidad**: Al no existir esta capa de aislamiento inherente a la virtualización, un bug en el entorno de un único usuario podría comprometer la integridad de la sesión para todos los operadores conectados.

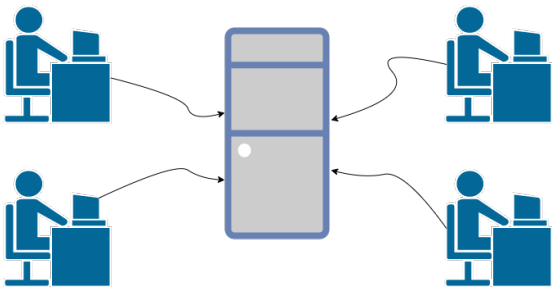

Figura 2.1: *Multiseating*.

2.2.2 Múltiples usuarios en área local

Este esquema (mostrado en la Figura 2.2) está formado normalmente por un mainframe y terminales, fue empleado ampliamente en el pasado en entornos profesionales [5]. La principal razón para su descenso en popularidad fue la reducción de costos en las unidades de procesamiento [6], llegado al punto en que dotar a cada operador de un equipo independiente resultó viable económicamente. Hoy día, la motivación para aplicar esta estructura puede tener más sentido desde un punto de vista ecológico que económico.

Este esquema resulta heterogéneo en lo que a software se refiere. Usualmente, se emplea una solución compleja para el mainframe sobre el que se conectan clientes genéricos mediante emuladores de terminal. El fabricante más reconocido de este tipo de equipos es IBM, con modelos como el IBM Serie Z.

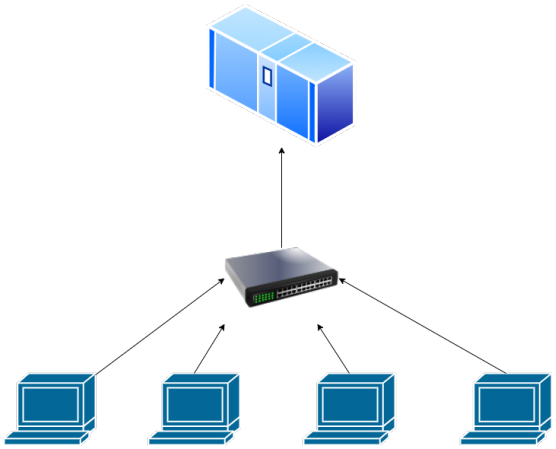

Figura 2.2: Conexión entre terminales y un mainframe.

Una variación de este esquema sería el arranque en red con almacenamiento remoto, haciendo uso de tecnologías como Preboot eXecution Environment (PXE)[1]. Usualmente, esta técnica emplea servidores de dominio y almacenamiento distribuidos. Contamos con tecnologías como Clonezilla, de software libre, que nos permiten configurar el servidor de arranque.

Este esquema es muy eficiente y no presenta grandes problemas *per se*. Sin embargo hereda el punto único de fallo asociado al mainframe –aunque siendo un equipo especializado cuenta ya con Tolerancia a fallos por diseño–. En el caso del arranque en red, este problema no está presente.

2.2.3 Múltiples usuarios en WAN

Según este modelo los clientes no se situan en la misma red física. Este escenario (Figura 2.3) y el anterior se pueden equiparar por medio de técnicas de tunneling. Es, por ende, una extensión del esquema previo (Figura 2.2). Para implementar esta arquitectura es necesaria la habilitación de un mecanismo de cifrado y un sistema de *tracking* para IPs dinámicas. Herramientas como PulseSecure VPN nos permiten manejar la conexión entre el usuario y los recursos de red remotos.

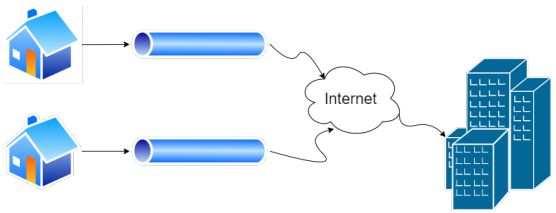

Figura 2.3: Múltiples usuarios en WAN.

[1] Entorno que permite la instalación de un sistema operativo a través de la red.

2.2.4 Virtualización en WAN

Aquí presentaremos múltiples soluciones, heterogéneas en su planteamiento; desde servicios de escritorio virtual remoto tales como VMWare Horizon hasta plataformas de streaming orientadas a videojuegos como Google Stadia. Conceptualmente se aproximan a nuestra propuesta, variando en protocolo de transmisión, propósito y arquitectura interna.

La principal diferencia entre el esquema multiusuario en WAN (equipos conectándose a recursos **compartidos y físicos** de una organización) y virtualización en WAN (usuarios accediendo a recursos **individuales y virtuales**), mostrado en la Figura 2.4, es el nivel de aislamiento entre operadores.

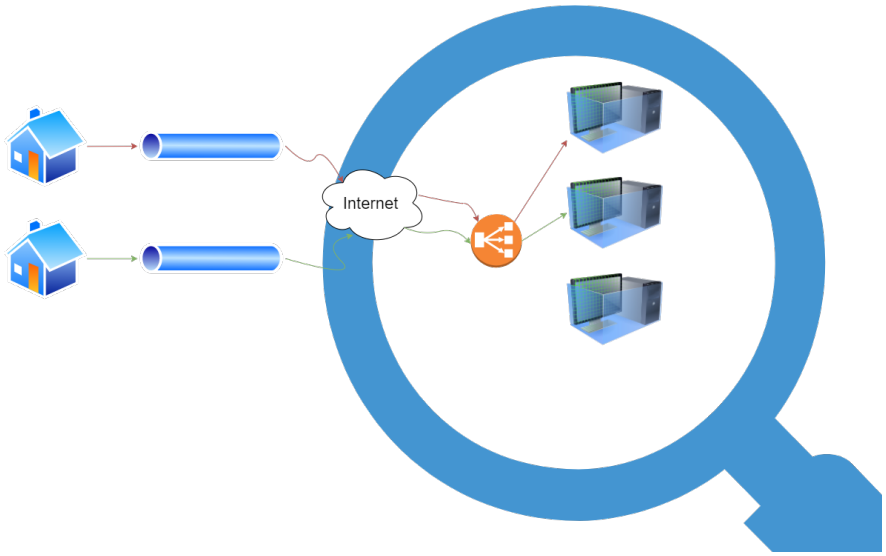

Figura 2.4: Virtualización en WAN.

2.2.5 Arquitectura Zero Client

La acotación más precisa que podemos hacer hasta ahora de Clobber en su vertiente de cliente es la del *cliente Thin o Zero*. Bajo este paradigma los equipos vienen dotados de un firmware capaz de iniciar conexiones de red y mostrar una interfaz interactiva. Suelen hacer un uso extensivo de las tecnologías Citrix. Dentro de esta modalidad encontramos las soluciones propuestas por Clearcube, 10Zig, Teradici PCoIP y los Dell Wyse Xenith. Desde el punto de vista de Clobber, estos equipos resultan redundantes, al no incluir la parte de *entrada/salida* sino únicamente conectores para los diferentes periféricos.

2.2.6 Arquitectura Webtop

El concepto de *webtop* se basa en el acceso a un espacio de trabajo a través del navegador. El servidor en esta arquitectura pone a disposición del usuario un entorno de escritorio con el que se interactua de manera análoga a un sistema tradicional. La primera herramienta en esta categoría de la que se tiene constancia fue desarrollada por la SCO[2] en el año *1993* bajo el nombre Tarantella. Desde mediados de los años *2000* han ido perdiendo relevancia. Es posible que en el futuro vean un nuevo resurgimiento de la mano de la virtualización; de hecho, en el último mes ha aparecido una nueva implementación basada en Docker desarrollada por linuxserver.io.

[2] Compañia Santa Cruz Operation

Capítulo 3

Herramientas y tecnología

Para el desarrollo de este trabajo, seleccionamos un repertorio de software que resultará familiar a los administradores de equipos *Linux*. Durante el presente capítulo justificaremos su elección y hablaremos de las alternativas sopesadas.

3.1 Lenguajes de programación

El planteamiento original de la aplicación consistía en un único servicio maestro en cada servidor. Por ejemplo, en el caso de las máquinas virtuales constaría de un device driver en *C* encargado de manejar cada petición mediante sockets de sistema. Sin embargo, cuando comenzó la fase de análisis determinamos que la mejor solución consistiría en emplear multitud de pequeños programas especializados. El motivo principal es la flexibilidad que aporta poder compaginar los diferentes comandos y herramientas propios de *Linux* con nuestro propio software, minimizando solapamientos entre su funcionalidad. Para aprovechar al máximo esta integración, decidimos emplear Shell Scripting como lenguaje principal. Nuestro software suele invocar a algún servicio *(Docker, TGT, etc)* y modifica el flujo de ejecución en función de la salida.

Para los *endpoints* de los servidores, debido a la experiencia previamente adquirida y a la seguridad que nos aporta poder reservar y liberar recursos de almacenamiento dinámicamente *(with)*, hemos decidido emplear *Python*.

En el caso de las interfaces web usaremos *HTML*, *CSS* y *JavaScript*, empleando PHP para el código encargado de la verificación de inicio de sesión. Finalmente hemos seleccionado MySQL como software para almacenar las credenciales de nuestros usuarios.

3.2 Bibliotecas y componentes

Acerca de las bibliotecas, para *Javascript* empleamos JQuery y JQuery-UI.

En el caso de los servidores *Python* hemos empleado Flask como framework para los *endpoints* debido a la versatilidad que presenta. Optamos Gunicorn –un servidor de aplicaciones–, al ser el que mayor grado de integración mantiene con Flask. Para los templates de cada página hacemos uso de Jinja2 –motor de plantillas web– ya que es opción por defecto en Flask. Por último empleamos Zipfile36 –librería de descompresión de ficheros zip en *Python*– para la descompresión de los comandos y SqliteDict –interfaz de interacción con bases de datos por medio de estructuras de tipo diccionario– por su sencillez y eficiencia.

En lo que a componentes empleados se refiere y debido a motivos temporales hemos empleado el template *Login Form V16 de Colorlib* [7] licenciado bajo *Creative Commons* para redistribución y edición *(CC BY 3.0)*, adaptado para invocar un script PHP de autenticación al presionar el botón de submit.

3.3 Tecnologías desplegadas

En los diferentes servidores se encuentran los siguientes progamas:

- **Gateway Router**

  El router (Figura 3.1) nos ofrecerá la posibilidad de configurar automáticamente nuevos equipos asociados a la interfaz de red interna por medio de del protocolo DHCP. El programa que hemos decidido emplear para esto es ISC dhcp server debido a la extensa documentación que existe en la red. Es importante puntualizar que en este momento las entradas asociadas a las máquinas de la red se definen de manera estática (siempre con la misma IP). Para las labores de servidor caché DNS hemos seleccionado la herramienta Bind9. Esta caché almacenará las peticiones de resolución de dominio más recientes en memoria para no tener así que ejecutar una búsqueda completa. En caso de no contar con una entrada requerida (navegamos, por ejemplo, a una página web por primera vez) empleará los servidores DNS de *Google* para resolver las peticiones. Finalmente las conexiones entre internet y el enrutador pasarán por un IPTables, el firewall que hemos seleccionado.

- **Servidor WEB**

  El servidor WEB (Figura 3.2) contiene un *stack* LAMP para el sitio web. Esta elección se debe a la naturaleza clásica de las páginas que mostraremos, consistiendo principalmente de paneles de documentación y un portal para el inicio de sesión. El *stack lamp* se compone de:

  – Apache: en este caso su segunda versión como servidor web. Esto es, el programa encargado de procesar las peticiones HTTP y servir los recursos adecuados.

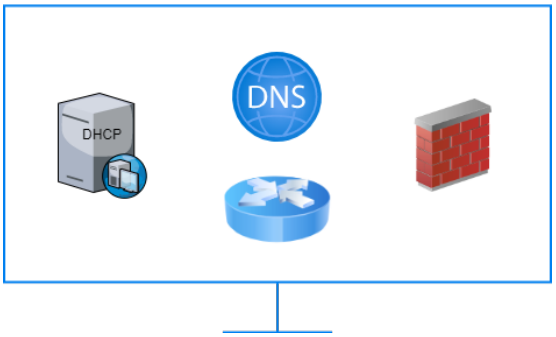

Figura 3.1: Software presente en el router

- MySQL: como servidor y administrador de bases de datos. Es el software que emplearemos para almacenar los datos de los perfiles de usuario asi como eventual contenido del sitio web.
- PHP: un intérprete PHP es necesario cuando deseemos integrar programas en ese lenguaje dentro de nuestro sitio. Nosotros lo empleamos en el software encargado del inicio de sesión.

Por último y para facilitar el acceso remoto a la máquina instalaremos un servidor SSH con la herramienta openssh server.

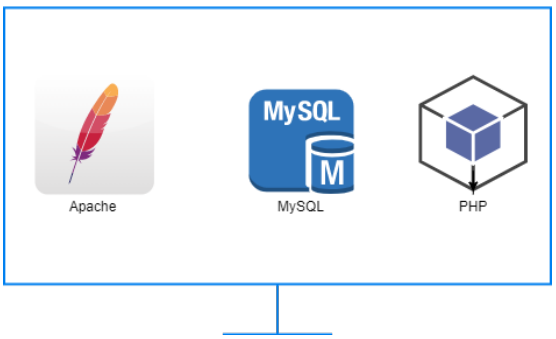

Figura 3.2: Software presente en el servidor web

- **Servidor de virtualización**

  El servidor de virtualización (Figura 3.3) se encarga de la contención y administración directa de las instancias Clobber. La tecnología de virtualización empleada es Docker. El software que permite conectar el almacenamiento en red mediante el protocolo iSCSI es open iscsi, tomando éste el rol de iniciador iSCSI. Por último instalamos también en esta máquina openssh server.

- **Servidor de almacenamiento**

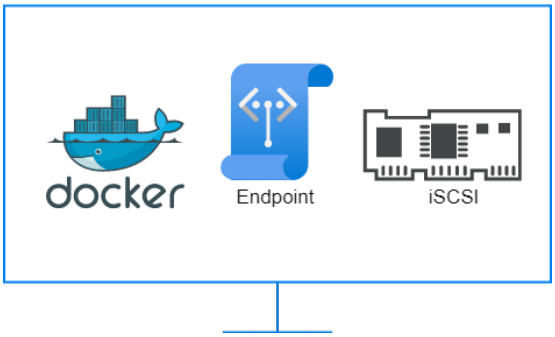

Figura 3.3: Software presente en el servidor de virtualización

Finalmente, el servidor de almacenamiento (Figura 3.4) empleará comandos y herramientas nativas de *LINUX* tales como *DD*, *mount*, *mkfs* para generar los dispositivos de almacenamiento, requiriendo únicamente de software para exponerlos en red. El protocolo empleado es iSCSI y la herramienta que lo implementa –esta vez con el rol de servidor– es TGT. Este servidor también posee una instalación de openssh server.

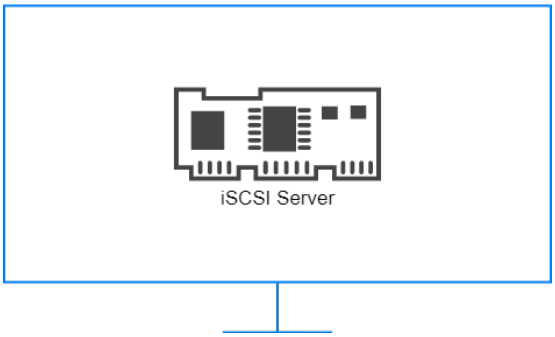

Figura 3.4: Software presente en el servidor de almacenamiento

3.4 Herramientas de desarrollo

Todo el software fue programado a mano utilizando el editor Atom en un equipo local. Levantamos el entorno mediante máquinas virtuales *Ubuntu* y *Debian* sobre VirtualBox. Las imágenes de las máquinas *Ubuntu* proceden de la página Osboxes.org. Este sitio nos las permite descargar de manera gratuita contando con un ámplio repertorio de distribuciones *Linux* preconfiguradas para su uso tanto con VirtualBox como con VMWare.

3.5 Herramientas de soporte

Para la descripción de las *APIs* empleamos la herramienta de edición de servicios RESTful Swagger siguiendo el estandar OpenAPI. En cuanto a la redacción de esta memoria empleamos el editor de documentos LaTeX Overleaf y las figuras han sido editadas mediante la herramienta GIMP. En la realización del proyecto hicimos uso de la herramienta de elaboración y seguimiento de proyectos MS Project de la empresa *Microsoft*. Todas las figuras fueron elaboradas en el editor de diagramas online drawio.

Capítulo 4

Metodología y planificación

Este capítulo contiene la justificación de la metodología empleada a lo largo del trabajo así como el desglose del *plan de proyecto* y una *simulación* de coste.

4.1 Metodología de desarrollo

Para comentar las diferentes metodologías empleadas, debemos realizar antes una clasificación de los diferentes programas elaborados para este trabajo.

- **Parser y motor Clobber (endpoints de la instancia Clobber)**: ambos elementos se encuentran integrados en el mismo programa *Python* y han sido desarrollados según una metodología iterativo incremental. Esto se debe a que el coste de implementación de un nuevo endpoint es independiente del tamaño del proyecto, lo que nos permite un desarrollo muy fluido incluso cuando los incrementos se encuentren espaciados a lo largo de la duración del trabajo.

- **Scripts del servidor de almacenamiento**: Los scripts del servidor de almacenamiento han seguido una metodología clásica estilo cascada debido a su escaso margen para incrementos ya que cumplen una función muy específica. En caso de requerirse nueva funcionalidad realizararemos un nuevo desarrollo en vez de modificar el actual.

- **Endpoints del servidor de virtualización**: Decidimos elaborar el software que interactúa con el servidor de virtualización siguiendo una metodología *en cascada* por causa de su escasa extensión.

- **Templates**: Los diferentes templates (programas que demuestran el uso de Clobber) siguieron un modelo iterativo incremental durante su desarrollo. Esto es importante ya que debemos asegurarnos de que las soluciones son sencillas de implementar para el usuario (recordemos que el usuario de Clobber *es un programador*).

- **Módulo de comandos**: en una primera revisión pensamos que sería interesante realizar un device driver. Su objetivo sería ejercer de usuario para el kernel; recibiendo las operaciones desde un sockets y devolviendo el *output* ya en formato JSON. Esta idea se justicaba ya que el modelo original empleaba un kernel propio, aprovechando la oportunidad para reemplazar el concepto de usuario tanto en el kernel del dispositivo como en el de la instancia Clobber. En vista de que finalmente emplearíamos una imagen de *Linux*, la complejidad del desarrollo y pruebas del device driver sumadas a la dificultad de recuperación ante errores exceden cualquier posible beneficio. Determinamos entonces desplazar el device driver a un futuro desarrollo junto con el propio kernel. En este caso lo sustituiremos por el *módulo de comandos*; éste ejecutará, bajo orden del *motor*, un comando o script mediante un nuevo proceso, empaquetando previamente sus descriptores de fichero 1 (stdout) y 2 (stderr), reportando el resultado de la ejecución en un JSON.

4.2 Planificación y seguimiento

La primera fase consistió en la elaboración de la línea base, esta une los camínos críticos del proyecto y da comienzo el día 22/03/2021 finalizando el 17/06/2021. Durante esta sección enunciaremos las tareas por las que pasó este trabajo, sus costes asociados y las diferentes estratégias de gestión de riesgos aplicadas.

4.2.1 Recursos

Hemos dimensionado el proyecto para un único desarrollador en el plazo de tres meses sin una inversión económica determinada.

El entorno de desarrollo se compone de un equipo de sobremesa dotado de un procesador *Ryzen 3800X* (8 núcleos) y *16GB* de memoria *RAM DDR4* a *3200MHz* corriendo *Ubuntu 20.04.2 LTS* como sistema operativo anfitrión. En este equipo además se despliegan tres imágenes con la misma versión de *Ubuntu* y una *Debian 4.19.0-16-amd64* mediante VirtualBox. El software que hemos instalado en cada una de las *VMs*, así como las herramientas de desarrollo se distribuyen de manera gratuita.

Para facilitar la labor de prueba y distribución de la versión alfa, asociamos la IP pública del equipo al subdominio *clobber.tretornesp.com*.

4.2.2 Gestión y planificación del proyecto

A la hora de planificar el proyecto se definió como métrica principal el esfuerzo en horas-hombre (HH) total requerido para su finalización.

Para el cálculo en HH computamos los créditos ECTS asociados a este trabajo. Un crédito ECTS es equiparable a apróximadamente 25 horas, por lo que realizamos la conversión siguiente:

$$f(creditos, hombres) = (creditos \times 25)/hombres$$

El resultado de aplicar la fórmula a 12 créditos y un único trabajador resulta en

$$f(12, 1) = (12 \times 25)/1 = 300hh$$

Aplicando este resultado a la fecha de entrega del proyecto *23 de junio* y al calendario laborar del trabajador, pudimos calcular la única variable restante: la fecha de inicio del proyecto.

Este acercamiento *inverso* resultó posible ya que contábamos desde un primer momento con una lista de funcionalidades que podrían entrar y salir del proyecto sin afectar a la consecución de nuestros objetivos y que nos aportó una enorme flexibilidad.

Los siguientes pasos a seguir consistieron en descomponer estas tareas lo máximo posible, identificar posibles dependencias entre ellas y elaborar la estimación de cada una. El resultado, una vez modelado en la herramienta MS Project, conformó el plan de proyecto.

El extracto de estas tareas junto a su estimación se muestra a continuación: [1]

- **Tarea 1: Recabar Información**

  Investigación del estado del arte, selección de tecnologías para el desarrollo y valoración del entorno a desplegar.

- **Tareas 2: Toma de requisitos**

  Elaboración de la lista de funcionalidades que la aplicación debe ofrecer. La mayor parte de esta lista se encontraba elaborada llegado el punto de realizar la tarea, por lo que estas horas fueron asignadas a la redacción de documentación.

- **Tarea 3: Elaboración anteproyecto**

  El anteproyecto conforma la documentación previa a la aprobación del proyecto.

- **Tareas 4 y 5: Análisis conjunto y casos de prueba**

  El programa no presenta dependencias entre los diferentes módulos (almacenamiento, virtualización y web). Gracias a esta particularidad las fases de diseño implementación y llevamos a cabo las pruebas para cada software de manera independiente. No obstante,

[1] Las tareas han sido agrupadas para reducir el tamaño del documento.

el análisis de la solución a nivel global resulta beneficioso a la hora de plantear un despliegue sencillo. Después de haber identificado que módulos deberíamos implementar, establecimos una lista de casos de prueba para cada uno.

- **Tarea 6: Diseño del API**

  El API como entidad conjunta resultó incoherente con el rediseño de la arquitectura (de *monolítica* a *modular*). Por lo que la asignación temporal se corresponde a la suma del tiempo invertido elaborando y documentando los diferentes *endpoints*.

- **Tareas 7, 8, 9 y 10: Despliegues**

  Las tareas de despliegue consisten en la creación de las diferentes máquinas virtuales incluyendo instalación, configuración y prueba del software que se ejecutará en cada una. En este caso la mayor parte del tiempo fue destinada a investigar las diferentes opciones de configuración para los programas, siendo la instalación relativamente rápida.

- **Tareas 11 14 y 17: Fase de diseño**

  El diseño comienza inmediatamente después del despliegue de entorno y precede a la implementación y prueba de cada módulo. El tiempo dedicado fue de unas 6 horas por cada servidor. Destinamos la mitad a rediseñar la arquitectura del software para las instancias Clobber.

- **Tareas 12, 15 y 18: Fase de implementación**

  La fase implementación no incluye la programación web (que corresponde a las tareas *20 y 21*, so no solamente los endpoints de los diferentes servidores y el software que interactua con el servicio iSCSI.

- **Tareas 13, 16 y 19: Fase de pruebas**

  A medida que íbamos considerando cada módulo como implementado, ejecutamos las pruebas diseñadas en la *Tarea 5*. Una particularidad de la programación cuando tratamos con lenguajes interpretados es la ausencia, en muchos casos, de *crashes*[2]. La mayoría de errores detectados degeneraban en salidas incompletas o nulas. La solución pasó en la mayoría de las ocasiones por *extender la funcionalidad* para soportar entradas incompletas mediante valores por defecto que modifican este comportamiento erróneo. *Algo de lo que nos sentimos orgullosos es de contar con una complejidad ciclomática extremadamente baja (métrica de complejidad lógica).*

- **Tarea 20: Elaboración del portal web**

[2] Error que aborta un programa.

La selección del *template* para el formulario de inicio de sesión –resultó complicado encontrar uno con licencias compatibles con el proyecto– así como el *script* *PHP* se demoraron debido a la baja experiencia previa con el *stack*.

- **Tarea 21: Elaboración de los ejemplos**

  Una grata sorpresa fue la velocidad con la que logramos llevar a cabo los ejemplos de implementación de la solución; la mayoría del tiempo lo dedicamos además a la edición del diseño CSS, pudiendo realizar la integración con el servicio en un breve lapso de tiempo.

- **Tarea 22: Elaboración de la memoria**

Escribimos este apartado en la última etapa del documento. La redacción de la memoria final ocupó un total de 6 horas diarias durante un periodo de dos semanas.

4.2.3 Resumen de tareas y estimación

El resumen de las tareas ordenadas se puede consultar en la Tabla 4.1:

La relación entre tareas y sus dependencias se pueden consultar en diagrama de Gantt presente en la Figura 4.1

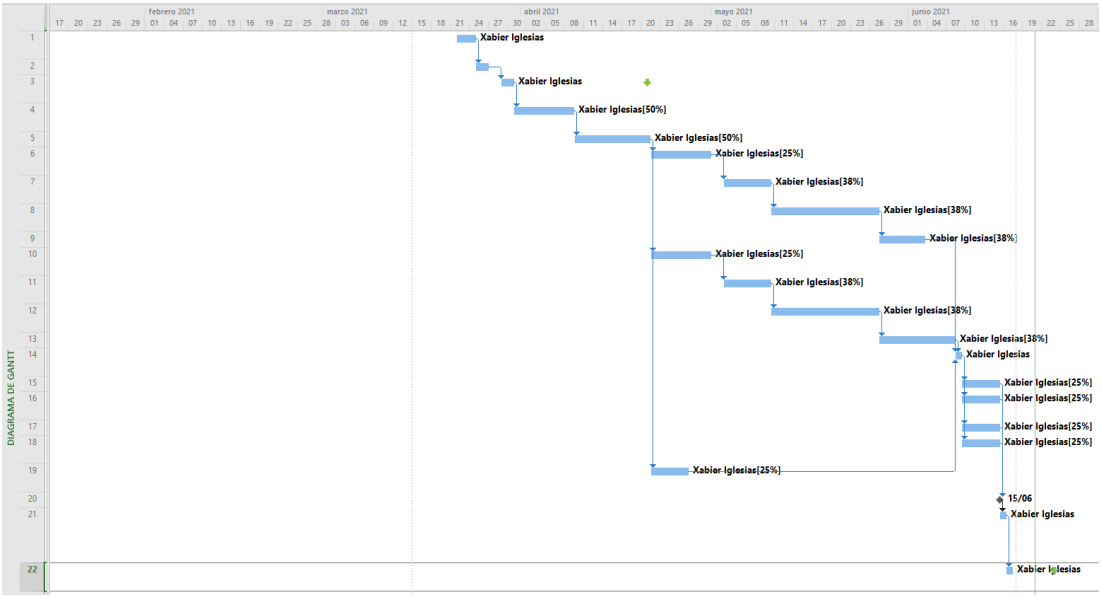

Figura 4.1: Diagrama de Gantt del proyecto

Nº	Tarea	Tiempo Estimado	Tiempo Real
1	Recabar Información	24	32
2	Toma de requisitos	16	4
3	Elaboración anteproyecto	16	12
4	Análisis conjunto	24	32
5	Casos de prueba	8	24
6	Diseño del API	32	20
7	Despliegue router	8	4
8	Despliegue servidor web	8	4
9	Despliegue servidor virtualización	8	16
10	Despliegue servidor almacenamiento	8	24
11	Diseño almacenamiento	6	8
12	Implementación almacenameinto	16	14
13	Pruebas almacenamiento	4	5
14	Diseño virtualización	6	8
15	Implentación virtualización	16	14
16	Pruebas virtualización	4	5
17	Diseño web	6	8
18	Implemetación web	16	14
19	Pruebas web	4	5
20	Elaboración portal WEB	8	16
21	Elaboración ejemplos	24	16
22	Elaboración memoria	64	60
TOTAL		**326**	**345**

Tabla 4.1: Resumen de tareas del proyecto

Rol	N. Recursos	Tarifa horaria
Analista	1	80
Administrador de Sistemas	2	40
Diseñador	2	60
Desarrollador	4	40
Ingeniero de pruebas	2	60

Tabla 4.2: Resumen de recursos

4.2.4 Costes

Para la elaboración del informe de costes haremos el ejercicio de imaginar que hemos subcontratado para el trabajo a un equipo formado por los recursos mencionados en la Tabla 4.2

Las tareas se dividen de la siguiente manera:

- T1, T2, T3, T4 y T22: Analista
- T6, T11, T14 y T17: Diseñadores
- T12, T15, T18, T20 y T21: Desarrolladores
- T5, T13, T16 y T19: Ingenieros de pruebas
- T7, T8, T9 y T10: Administr. Sistemas

De aquí se obtienen las siguientes horas estimadas por rol:

- Analista: $(T1 + T2 + T3 + T4 + T22) = (24 + 16 + 16 + 24 + 64) = 144h$
- Diseñadores: $(T6 + T11 + T14 + T17) = (32 + 6 + 6 + 6) = 50h$
- Desarrolladores: $(T12 + T15 + T18 + T20 + T21) = (16 + 16 + 16 + 8 + 24) = 80h$
- Ingenieros de pruebas: $(T5 + T13 + T16 + T19) = (8 + 4 + 4 + 4) = 20h$
- Administr. Sistemas: $(T7 + T8 + T9 + T10) = (8 + 8 + 8 + 8) = 32h$

De la suma ponderada según los precios horarios obtenemos el siguiente coste estimado:

$$144h \times 80eur/h + 50h \times 60eur/h + 80h \times 40eur/h + 20h \times 60eur/h + 32h \times 40eur/h =$$

$$(11520 + 3000 + 3200 + 1200 + 1280)euros = 20200eur$$

Con lo que el presupuesto estimado de nuestro proyecto, asumiendo que no existen ni costes fijos ni asociados a la infraestructura sería de *20200 euros*.

Si repetimos la operación con las horas reales:

$$140h \times 80eur/h + 44h \times 60eur/h + 74h \times 40eur/h + 39h \times 60eur/h + 48h \times 40eur/h =$$

$$(11200 + 2640 + 2960 + 2340 + 1920)euros = 21060eur$$

El resultado final es de **21060 euros**, lo que corresponde a una desviación de **19hh** y un sobrecoste asociado de **860 euros**

4.2.5 Gestión de riesgos

Debido a que el costo por hora real de los recursos humanos es de 0.00€ y a que no existen costes fijos ni variables apreciables asociados a la infraestructura de desarrollo, el único criterio para elaborar un plan de gestión de riesgos es su dimensión temporal.

La principal herramienta de la que dispusimos para detección de alteraciones con respecto a la línea base fue la revisión periódica.

En segundo lugar redactamos una lista de funciones opcionales que sirvieran de comodín, pudiendo incluirlas o descartarlas sin perjuicio alguno para los objetivos del proyecto.

El plan de contingencia principal sería, en el caso de una desviación sobre la línea base, desplazar elementos desde la lista de funciones opcionales hacia los desarrollos futuros en caso de habernos retrasado, o a la inversa en caso de un adelanto.

Sobre la estimación

Juan Ares estaba en lo cierto cuando hizo hincapié en la estimación como una de las labores más difíciles y que más experiencia requieren por parte de un ingeniero de software.

Capítulo 5

Análisis

La fase de análisis de este proyecto resultó tanto la más sencilla como la más importante. La facilidad que proporciona el hecho de que la especificación parta de uno mismo, permitió avanzar rápidamente durante la toma de requisitos. Siendo prácticamente un problema compartido entre análisis y diseño ya que podemos buscar los requisitos a un nivel más bajo del que un cliente externo nos podría especificar.

Es importante realzar que Clobber es una herramienta para desarrolladores, no para usuarios comunes. Este hecho modifica profundamente el esquema de actores y acciones clásico de una aplicación.

5.1 Análisis de requisitos

El análisis de requisitos comenzó con la demostración principal, el escritorio. Su enunciado es el siguiente:

Realizar una pantalla semejante al escritorio clásico de un sistema operativo de uso común, de interacción mediante un navegador web, que emplee toda la tecnología de Clobber que sea posible.

Para este ejemplo tomamos como referencia el escritorio presente en el sistema operativo Windows 10. Identificando sus elementos tanto visuales como funcionales, obtuvimos la siguiente lista de funciones:

- Buscar rápidamente ficheros por nombre.
- Mostrar la fecha y hora del sistema.
- Mostrar el usuario actual.
- Acciones básicas de seguridad (cambio de contraseña y desconexión).
- Enlace con cualquier otra aplicación integrable (acceso a otros ejemplos).

Las funciones expuestas por nuestro servicio para el programador parten de esta lista y quedarán recogidas en apartados posteriores. Desde el punto de vista de la infraestructura interna, hemos partido del estudio de varias posibles arquitecturas con el fin de identificar los equipos que formarían parte de nuestra red en primera instancia, para posteriormente enumerar las funciones de las que debe encargarse cada uno.

Para obtener los requisitos no funcionales, hemos tomado el papel de dos supuestas empresas que emplean tecnología Clobber: 1) Una pequeña empresa que cuenta con recursos humanos limitados y 2) Una gran empresa cuyos servicios tengan un volumen de peticiones alto.

5.1.1 Actores

Nuestros actores, en el caso de la aplicación global son los programadores que en un momento determinado consumen nuestras APIs genéricas o añaden las suyas propias. Éstos son los dos casos de uso principales.

Para cada módulo interno contamos con un único actor que genera peticiones de acción:

- **Servicio iSCSI** El actor es el iniciador iSCSI situado en el servidor de virtualización.
- **Servicio de virtualización** El actor es el código del servicio web que inicia las peticiones de administración concretas.
- **Servicio Clobber** El actor es el código de frontend que ejecuta la petición.
- **Servicio Web** El actor en este caso sí sería un usuario final para el CU de inicio de sesión.

La interacción de los actores con el sistema se ejemplifica en la Figura 5.1 para una acción genérica y en la Figura 5.2 para el caso específico del inicio de sesión.

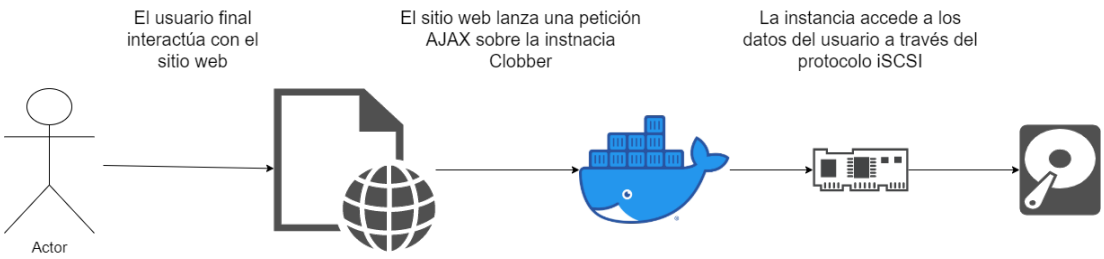

Figura 5.1: Ejemplo de interacción entre actores en un caso general

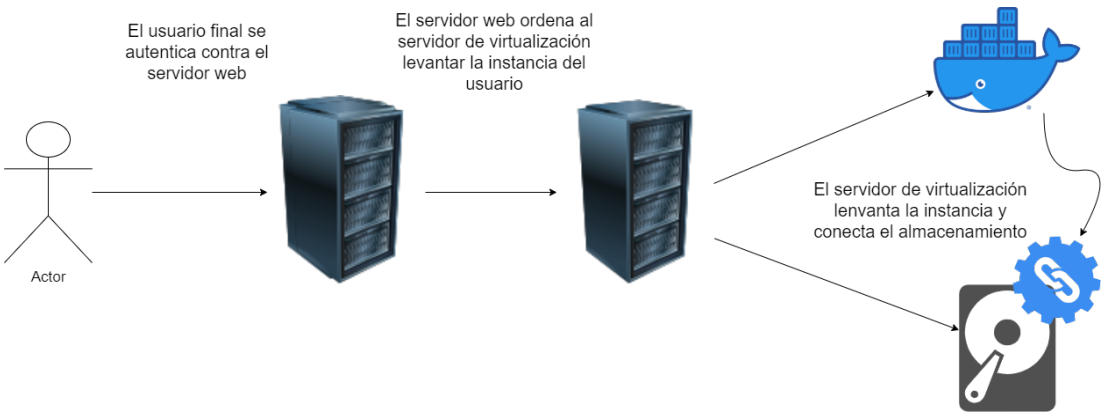

Figura 5.2: Ejemplo de interacción entre actores durante el inicio de una sesión

5.1.2 Requisitos funcionales

Los requisitos funcionales globales (de la solución como caja negra) serían:

- Enviar y recibir comandos de un servidor virtual.
- Añadir comandos arbitrarios al servidor.
- Administrar los servidores virtuales de manera remota.
- Administrar los usuarios asociados a cada servidor (Agregar, Eliminar, Modificar).

Requisitos funcionales del servicio iSCSI

- Crear y eliminar discos duros aprovisionado desde un fichero.
- Añadir y eliminar *targets iSCSI* asociados a los discos y denotados por una nomenclatura arbitraria.

Requisitos funcionales del servicio de virtualización

- Agregar y eliminar contenedores Docker con la imagen y el nombre especificados.
- Obtener datos de configuración de un contenedor tales como la IP o sus puertos abiertos.
- Montar almacenamiento en red proveniente del servicio iSCSI y transmitirlo a los contenedores virtuales.

Requisitos funcionales del servicio Clobber

- Exponer una serie de endpoints.
- Ejecutar comandos arbitrarios y devolver la salida.
- Filtrar la salida de los comandos.
- Añadir o eliminar endpoints arbitrarios.

Requisitos funcionales del servicio Web

- Agregar, eliminar y modificar usuarios.
- Iniciar y cerrar sesión de un usuario.
- Redirigir al template asociado a cada usuario.

5.1.3 Requisitos no funcionales

Los requisitos no funcionales generales son:

- Soporte para un amplio número de contenedores por servidor de virtualización.
- Tiempos de respuesta de la infraestructura bajos.
- Tolerancia a fallos.
- Escalabilidad horizontal y vertical.
- Definición de APIs según estandar OpenAPI 3.0.
- Empleo de JSON como formato de recepción de datos.
- Modularidad (Sustitución sencilla de los subsistemas).
- Segregación de servicios en múltiples servidores.

Requisitos no funcionales del servicio iSCSI

- Independencia del sistema de ficheros.
- Tiempo breve en la creación de los ficheros de disco.
- Cantidad no inferior de *targets* con respecto a los contenedores máximos por servidor de virtualización.

Requisitos no funcionales del servicio de virtualización

- Tiempos razonables en la creación de contenedores.
- Aislamiento contenedor/anfitrión y contenedor/contenedor.

Requisitos no funcionales del servicio Clobber

- Operaciones que no generen estado más allá de los ficheros en red cuando sea posible.
- Uso de ficheros comprimidos para agregar los comandos.

Requisitos no funcionales del servicio Web

- Filtrado de peticiones inseguras (magic quotes, SQL injection, etc).
- Cifrado de las credenciales.
- Comunicaciones cifradas entre el servidor web y los servidores de virtualización/almacenamiento.

5.2 Arquitectura

Entramos ahora en el núcleo de este trabajo. La arquitectura de la *solución* es parte fundamental de la *solución* en sí misma. De las decisiones tomadas en este apartado derivamos muchas de las características que diferencian el resultado final de las vistas en el análisis del estado del arte.

5.2.1 Arquitectura global

Lo primero que tuvimos en cuenta para elaborar la arquitectura son los diferentes tipos de servicios prestados; es razonable segregar desde un primer momento la capa de virtualización (anfitrión para los contenedores) de la capa de almacenamiento para aumentar su modularidad y eficiencia. Por seguridad, resulta interesante segregar también el servidor web del resto de las capas (se podría introducir un firewall intermedio con mayor facilidad). Una vez determinadas estas tres capas, por sencillez y limitación física decidimos realizar una única red, aislada mediante un firewall que solo permitirá comunicación por puertos determinados con los diferentes servicios.

A pesar de quedar relegado a un trabajo futuro, uno de los puntos más importartes de la solución es la escalabilidad horizontal en las capas de virtualización y almacenamiento. Debido a esto, orientamos la arquitectura hacia este objetivo.

La arquitectura global de Clobber se puede visualizar en la Figura 5.3

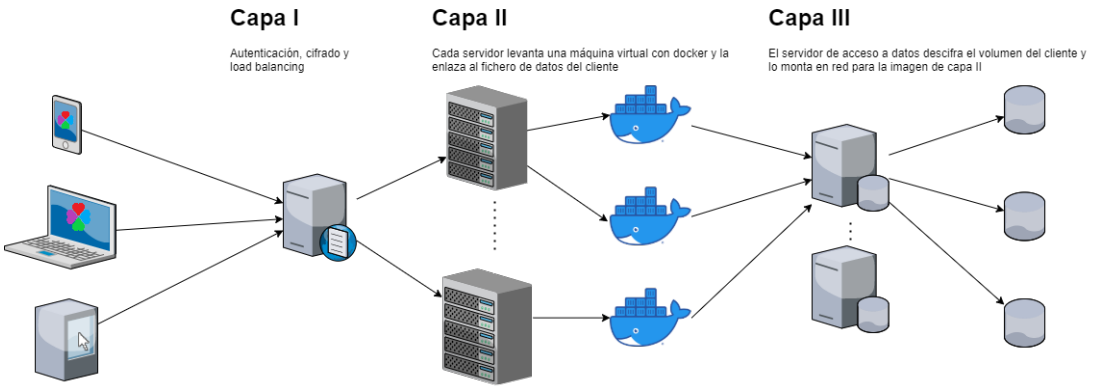

Figura 5.3: Arquitectura general de Clobber

5.2.2 Servicio de almacenamiento

El equipo de almacenamiento consta de un servidor iSCSI accesible en red local que expondrá un *target* por cada disco de usuario. El nombre del *target* seguirá el formato regular *iqn.com.clobber:DISCO*, siendo *DISCO* el identificador único de la unidad de almacenamiento asociada. Para asegurar su singularidad emplearemos la clave primaria de la tabla de usuarios del servicio web (en este caso el nombre de usuario).

Para su administración cuenta con un servidor SSH con política nopasswd (solo permitimos el login mediante clave pública) a través del que ejecutamos remotamente dos scripts. Uno para la creación de un nuevo disco y su *target*, y otro para la tarea de limpieza.

5.2.3 Servicio de virtualización

El servicio de virtualización cuenta con una Dockerfile base con la que genera la imagen una única vez, posteriormente mediante el mismo mecanismo de scripts por SSH permite levantar y eliminar un nuevo contenedor, limpiar cada instancia y obtener información de cada máquina. También existe un script que permite introducir en lista blanca la IP pública del cliente durante un tiempo máximo determinado.

5.2.4 Servicio web

La arquitectura de este servidor requiere de un sitio web clásico trabajando en combinación con un administrador maestro para los servidores internos. Por ejemplo, debe enviar la orden de levantar una máquina cuando un nuevo usuario decida conectarse.

Contamos con un servicio LAMP (*Linux*, *Apache*, *MySQL*, *PHP*; desplegado a mano herramienta por herramienta) manejando peticiones HTTP provenientes de usuarios finales y con

otro servicio, que mediante *endpoints* en localhost, ejecutan los diferentes scripts por SSH que hemos mencionado previamente.

En un desarrollo futuro, sería conveniente desplazar el servicio de administración a un nuevo servidor, para el que existen localizaciones razonables:

- Misma capa en paralelo al servidor web
- Nueva capa entre el servidor web y el servidor de virtualización.

La segunda opción permitiría intruducir cierta elasticidad en la capa de virtualización así como un agrupamiento geográfico de los servidores más flexible.

La interacción entre estos servicios queda plasmada en la Figura 5.4

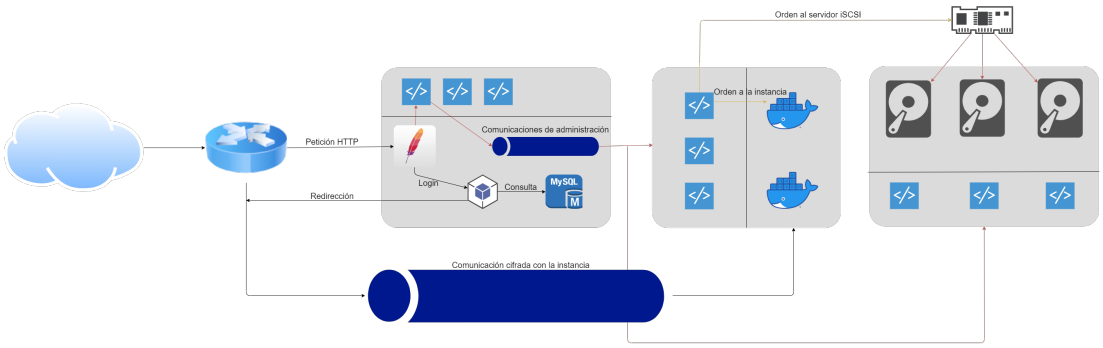

Figura 5.4: Ejemplo de interacción entre servicios

5.2.5 Consideraciones generales a la arquitectura del futuro cliente

Existe un pequeño prototipo en ensamblador x86 con el *stack* *LwIP*. En su descripción se tienen en cuenta las siguientes particularidades:

- Kernel monolítico.
- Sin sistema de ficheros, emplea hashes como identificadores de fichero.
- Soporte para un único usuario con permisos totales.
- Motor nativo de cifrado acelerado (Equivalente a Intel SHA Extensions).
- Device drivers mediante el modelo IOCTL de Linux (Para facilitar el desarrollo).

Por lo demás, el dispositivo debería ser capaz de ejecutar un navegador con la máxima fluidez, reproducir y capturar vídeo de alta calidad en tiempo real y manejar dispositivos de almacenamiento externos con diferentes sistemas de ficheros.

Capítulo 6

Desarrollo

El proceso de desarrollo en este proyecto se ha visto influenciado por su planificación, realizando un plan de despliegue de infraestructuras anterior a la programación del software. El despliegue y configuración de las máquinas comenzó con el *router*, seguido del *servidor web*; a continuación el *servidor de virtualización* y finalmente el *servidor de almacenamiento*. La fase de desarrollo siguió el camino opuesto.

Durante este capítulo hablaremos de los pasos necesarios con el fin de desplegar cada uno de los diferentes servidores, para posteriormente hablar del desarrollo de los módulos, tratando de explicar los puntos del código que pudieran tener alguna implicación importante en *eficiencia, seguridad o modularidad.* El orden seguido en esta sección se corresponde con la evolución cronológica del proyecto, comenzando por el despliegue del router y finalizando con el software del servidor web.

El resultado de este capítulo se puede consultar en el repositorio *GitHub* del proyecto[1].

6.1 Despliegue de servicios

En esta sección enumeraremos los pasos seguidos para levantar los diferentes servicios que conforman Clobber, estos servicios quedan reflejados en la Figura 6.1

6.1.1 Gateway router (10.10.24.1)

El router es el único dispositivo accesible directamente desde internet; contiene un firewall que redirige el puerto *80* hacia la máquina web. Así mismo contiene un servidor DHCP y caché DNS propia de la propia infraestructura.

[1] https://github.com/tretornesp/clobber

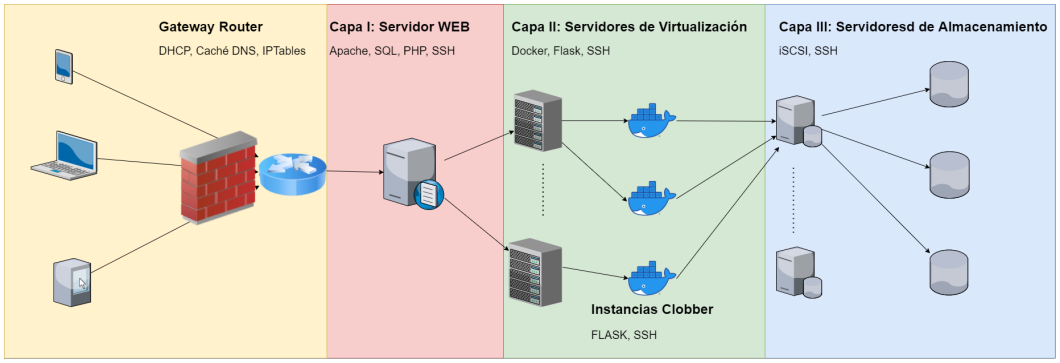

Figura 6.1: Diagrama de servidores y servicios

Configuración de la máquina virtual

Para esta máquina y habiendo estudiado la asignatura *Administración de Infraestructuras Informáticas*, hemos seleccionado una imagen de *Debian* dotada de dos interfaces de red:

- Red interna (clobber), adaptador: *enp0s3*.
- Adaptador puente, adaptador: *enp0s8*.

Este sistema estará provisto de 1GB de memoria RAM y un único núcleo de CPU.

Configuración como router

Permitimos redirección de trafico IPv4 empleando el comando:

```
sysctl net.ipv4.conf.eth0.forwarding=1
```

Configuración del servidor DHCP

Como servidor DHCP emplearemos ISC dhcp server, se instala mediante el comando:

```
apt install isc-dhcp-server
```

En el fichero `/etc/default/isc-dhcp-server` añadimos:

```
INTERFACESv4="enp0s3"
```

En el fichero `/etc/dhcp/dhcpd.conf` añadimos tres direcciones estáticas:

- **10.10.24.2** Asociada a la MAC del servidor de virtualización.

- **10.10.24.3** Asociada a la MAC del servidor de almacenamiento.
- **10.10.24.4** Asociada a la MAC del servidor WEB.

Configuración del servidor DNS

Instalamos Bind9:

```
apt install bind9 bind9utils
```

Modificamos `/etc/bind/named.conf.options` para incluir la opción forward only y especificando los servidores primarios de nuestra preferencia.

Configuración de IPTables

Se redirige el puerto *80* local a la IP del servidor web [8], el puerto *9420* al servidor de virtualización y los puertos *5000 a 6000* al servidor de virtualización [9, 10].

```
1 iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-
    destination 10.10.24.4:80
2 iptables -t nat -A PREROUTING -p tcp --dport 9420 -j DNAT --to-
    destination 10.10.24.2:9420
3 iptables -t nat -A PREROUTING -p tcp --dport 5000:6000 -j DNAT --to-
    destination 10.10.24.2:5000-6000
4 iptables -t nat -A POSTROUTING -j MASQUERADE
5 service iptables save
```

6.1.2 Servidor WEB (10.10.24.4)

El servidor web utilizará Apache, MySQL y PHP para servir las páginas; por otro lado permitirá acceso remoto para labores de administración mediante SSH. En esta sección desglosaremos el proceso de despliegue de esta máquina:

Configuración de máquina virtual

Esta máquina empleará una imagen de *Ubuntu* proveniente de Osboxes.org. La configuración incluye 2GB de memoria, un núcleo de CPU y una interfaz de red conectada a la red interna (clobber), con nombre de adaptador *enp0s3*.

Instalación de MySQL

El servicio MySQL contendrá los datos de los usuarios, lo instalaremos y configuramos la contraseña para el root:

```
apt install mysql-server
mysql\_secure\_installation
```

El progama desplegará en ese momento un menú de selección, optaremos por "validar contraseña" y "seguridad fuerte" para inmediatamente después introducir la clave deseada por duplicado.

Una vez configurado podemos ejecutar el siguiente script:

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `username` varchar(80) NOT NULL,
  `password` varchar(80) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Mediante la orden:

```
mysql -u root -p [contraseña] < script.sql
```

En ese momento ya podremos crear un usuario de pruebas mediante sentencias insert.

Instalación de Apache2 y PHP

El despliegue de Apache es relativamente sencillo: sólamente tenemos que realizar la instalación [11] y modificar la configuración del firewall para que este permita la conexión a través de sus puertos:

```
apt install apache2
ufw allow 'Apache'
```

Para ejecutar código PHP necesitamos:

```
apt install php libapache2-mod-php # Instalamos el software
a2enmod php                        # Añadimos php a la carpeta mods-
    enabled de apache
systemctl restart apache2          # Reinciamos el servicio
```

Por último copiamos el contenido de nuestro sitio web a la carpeta `/var/www/html/`

6.1.3 Servidor de Virtualización (10.10.24.2)

Este servidor contendrá las máquinas Docker y montará el almacenamiento en red.

Configuración de la VM

Empleamos la misma imagen de Ubuntu procedente de Osboxes.org, en este caso con *4GB* de memoria y un núcleo de CPU, con un adaptador de red conectado a red interna (clobber; *enp0s3*).

Instalación de Docker

El proceso de instalación de Docker en *Ubuntu 20.04* consta de los siguientes pasos:

```
1 apt install apt-transport-https ca-certificates curl \
2             software-properties-common
3 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
4            sudo apt-key add - # añadimos las claves GPG
5 add-apt-repository \
6   "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal
    stable" # agregamos el repositorio de Docker
7 apt update # actualizamos la BBDD de apt con los repositorios de
    docker
8 apt install docker-ce #Instalamos docker
```

Instalación del cliente iSCSI

La instalación de open iscsi se ejecuta con el siguiente comando:

```
apt install open-iscsi
```

6.1.4 Servidor de Almacenamiento (10.10.24.3)

El servidor de almacenamiento requiere únicamente de un administrador iSCSI. En este caso hemos seleccionado la herramienta TGT.

Instalación de TGT

TGT se puede instalar con el comando:

```
apt install tgt
```

6.1.5 Configuración de red común a los servidores

Existen dos parámetros de configuración que debemos ajustar en todos los servidores:

- El servidor DNS.
- El acceso por SSH.

Configuración del resolver DNS

Para obligar a los servidores a utilizar el servidor DNS situado en el router, modificaremos el fichero `/etc/resolv.conf` especificando bajo el apartado nameserver la IP del router *10.10.24.1*.

Configuración del acceso SSH

El acceso SSH se realizará desde el servidor web hacia los servidores de almacenamiento y virtualización por medio de certificados. Para esta labor debemos copiar los certificados a cada máquina. Para esto ejecutaremos los comandos:

```
1 ssh-keygen
2 sh-copy-id -i ~/.ssh/id\rsa.pub root@10.10.24.2
3 ssh-copy-id -i ~/.ssh/id\rsa.pub root@10.10.24.3
```

Posteriormente, en los servidores de almacenamiento y virtualización, editaremos el fichero `/etc/ssh/sshd_config` especificando el valor *without-password* para el campo PermitRootLogin.

Para finalizar reiniciamos el servicio con *systemctl restart sshd*.

Las diferentes máquinas, una vez desplegadas, se verían de manera semejante a la Figura 6.2

6.2 Software de los servidores

En este apartado cubriremos los puntos fundamentales del software que se ejecuta en cada uno de los servidores.

6.2.1 Software del servidor de almacenamiento

El servidor de almacenamiento cuenta con dos scripts para generar y eliminar *targets iSCSI*. En este contexto un *target* es una agrupación de unidades de almacenamiento, estas unidades lógicas reciben la denominación de LUN. Cuando deseamos acceder a un recurso de almacenamiento mediante el protocolo iSCSI el iniciador iSCSI (cliente de una conexión iSCSI) lanza una solicitud asociada al target que contiene al recuro. Tanto los *iniciadores* como los *targets* se identifican empleando una nomenclatura denominada *iSCSI Qualified Name* (IQN), el formato estandar es:

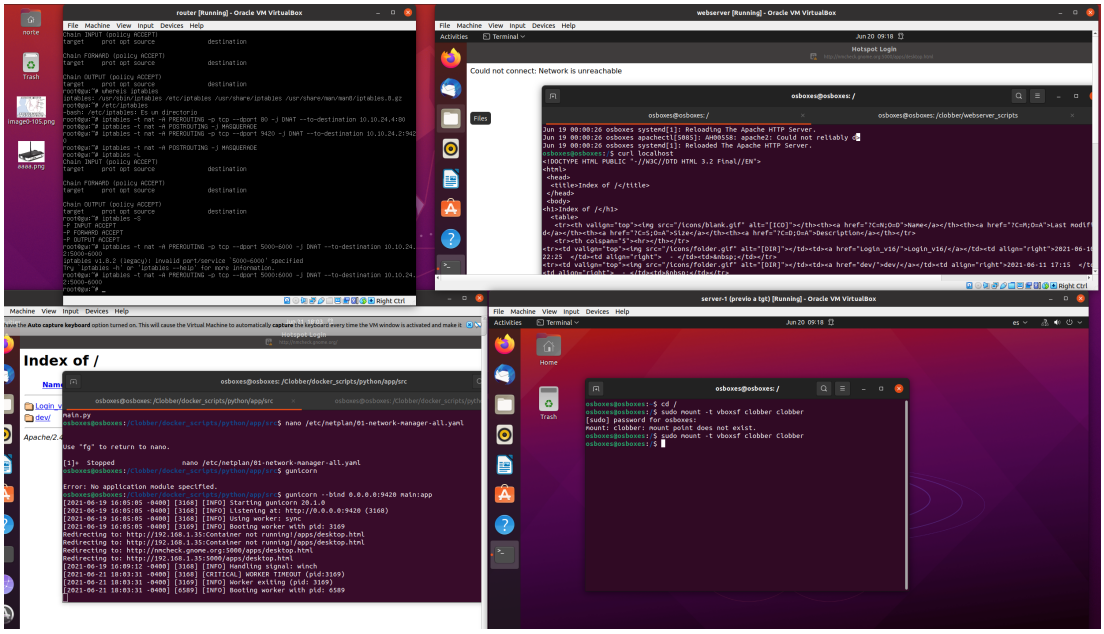

Figura 6.2: Captura de pantalla de los servidores desplegados

iqn.yyyy-mm.nombre-autoridad:unico donde:

- **yyyy-mm** es la fecha en la que se creó la autoridad que define el target.
- **nombre-autoridad** es el dominio de quien define el target.
- **unico** es un nombre único en todo el dominio.

De ahora en adelante haremos referencia a estos términos simplemente como: *target, iniciador, LUN* e *IQN*.

Nuevo target (new_target.sh)

Este script toma como parámetros el tamaño del disco y el nombre de usuario –que debe ser único– y devuelve via stdout el nombre del nuevo *target*. Hemos empleado la siguiente referencia para elaborarlo: [12].

El pseudocódigo de este script es el siguiente:

```
1 Copiamos n bytes de /dev/zero al fichero especificado
2 Creamos un sistema de ficheros ext4 sobre el fichero
3 Obtenemos el id del último target creado
4 Creamos un nuevo target con el nuevo id y el nombre iqn.com.clobber:
      nombre
5 Añadimos la unidad lógica asociada al fichero de almacenamiento
    creado previamente
```

```
6 Habilitamos el hooking global para nuevo target
7 Devolvemos el iqn del nuevo target
```

El tiempo de ejecución de este script depende del tamaño del disco especificado. Podría completarse en un período de tiempo constante restringiendo los tamaños de disco y empleando copias de ficheros previamente generados.

Eliminar target (delete_target.sh)

Este script elimina un target dado el nombre de usuario, así como el fichero de datos.

```
1 Extraemos el tid (\textit{target id}) a partir del nombre de usuario
2 Eliminamos el nodo asociado al tid
3 Eliminamos el fichero de datos con el nombre de usuario
```

Limpiar targets (cleanup.sh)

Este script elimina todos los targets y el almacenamiento asociado.

```
1 Extraemos el id del último target creado
2 Iteramos desde 1 hasta el tid obtenido previamente
3 Para cada número:
4     eliminamos el nodo asociado a la iteración actual
5 Vaciamos la carpeta que contiene todos los ficheros de datos
```

El tiempo de ejecución de este programa depende del número de targets iSCSI que existan en la máquina.

6.2.2 Software del servidor de virtualización

El servidor de virtualización contiene el software responsable de encender y apagar instancias Clobber, de la comunicación acerca de su estado y de la autorización de clientes.

Permitir conexión entre la instancia Docker y el cliente (allow.sh)

Este programa toma la IP pública del cliente y su nombre de usuario. Se basa en la siguiente documentacion: [13].

```
1 Permitimos conexiones desde la ip pública hacia el puerto de la
      instancia clobber
2 Programamos el bloqueo de las conexiones en un plazo determinado (
      timeout)
```

El sistema aquí expuesto no ofrece seguridad en un entorno real de producción, ya que:

- Cualquier equipo bajo la misma NAT de un cliente podría acceder al sistema.
- Solo se permite un cliente por NAT.
- Las IPs públicas rotan cada cierto tiempo.
- Se podrían inyectar comandos arbitrarios con paquetes que suplanten la IP de un cliente legítimo (mediante herramientas como Scapy).

Crear una nueva instancia (new_instance.sh)

Este programa se encarga de interactuar con Docker para levantar una nueva instancia basada en la imagen de *Clobber*. Toma como parámetro el nombre de usuario y devuelve la IP local de la nueva máquina. En caso de error muestra información relevante. Para la elaboración de este programa se ha consultado la siguiente documentación: [14, 15].

```
1 Si el contenedor está corriendo:
2     mostrar error y salir
3 Desconectamos el cliente iSCSI por si hubiese quedado abierto
4 Volvemos a conectar el cliente iSCSI al target y obtenemos el
      dispositivo asociado
5 Creamos una carpeta con el nombre de usuario y montamos el
      dispositivo en ella
6 Lanzamos la instancia Docker
7 Obtenemos y devolvemos la IP de la instancia
```

El comando para levantar la instancia Docker correspondiente a la linea 6 del pseudocódigo anterior es:

```
docker run -d -p 0.0.0.0:5000-6000:5000 -p
0.0.0.0:52022-53022:22 --name $1 --volume /fs/$1:/home/clobber/
cdata --volume /Clobber/python/app/src/:/server clobber
```

Donde estamos empleando la imagen denominada *clobber* para generar una instancia con el nombre del usuario, redirigiendo el puerto *5000* del contenedor a un puerto local en el rango *5000 a 6000* y el puerto *22* del contenedor al rango local *52022-53022*. Por último compartimos dos volúmenes con el contenedor:

- En la carpeta del contenedor `/home/clobber/cdata` situaremos el disco duro en red.

- En la carpeta `/server` almacenaremos el código de servidor que se ejecutará dentro de la instancia.

Se puede observar aquí que la instancia cuenta con dos puertos abiertos:

- **5000**: El servicio general de la instancia.
- **22**: Un servidor SSH para labores de prueba.

Eliminar una instancia (remove_instance.sh)

Este programa elimina una instancia dado el nombre de usuario y emplea comandos obtenidos en la siguiente referencia: [16].

```
Detenemos la instancia asociada al nombre de usuario
Eliminamos el contenedor asociado al nombre de usuario
Desconectamos el cliente iSCSI
Desmontamos el fichero compartido
Eliminamos la carpeta del usuario
```

Limpiar las instancias (cleanup.sh)

Este programa elimina todas las instancias activas del servidor de virtualización sin alterar su almacenamiento. Es útil para realizar el apagado o migración del servidor.

```
Paramos todas las instancias de la máquina
Eliminamos todos los contenedores
Desconectamos todos los clientes iSCSI
Para cada punto de montaje
    Desmontamos
Vaciamos la carpeta con los directorios de montaje
```

Obtener el puerto de servicio de una instancia (get_instance.sh)

Este programa devuelve el puerto que el servidor de virtualización está asociando a la instancia del usuario especificado:

```
Si el contenedor no está activo
    Mostrar un error y salir
Obtenemos los puertos activos de la instancia
Si el primer puerto está en el rango 5000-5999
    Devolvemos el puerto y salimos
Si el segundo puerto está en el rango 5000-5999
```

```
7       Devolvemos el puerto y salimos
8   Si no
9       Mostramos un error y salimos
```

Las comprobaciones asociadas al segundo puerto solo son necesarias si el puerto SSH está activo.

Endpoint: redireccionar a la instancia correcta main.py

Este programa emplea un endpoint Flask al cual suministraremos, mediante la URL, un nombre de usuario. A partir de este nombre efectuará una redirección hacia el puerto de enlace de su instancia asociada:

```
1 Obtenemos el puerto invocando a get instance
2 Extraemos el host de la petición entrante
3 Efectuamos una redirección hacia mismo host en el puerto de la
      instancia
```

Al respecto de la penúltima línea de este programa, como hemos visto en el script que lanza las instancias, el servidor de virtualización actúa como pasarela única hacia las máquinas empleando para ello redirección de puertos. Debido a esto, para desplazarse desde el servicio hacia una máquina solo hay que cambiar el puerto.

6.2.3 Software del servidor WEB

El servidor web permitirá además de su función principal de servir contenido, labores de administración centralizadas. Actualmente, permite la creación de instancias y realizar un borrado de usuarios.

Crear un nuevo usuario (new_user.sh)

Este programa se limita a ejecutar a través de una conexión SSH los scripts *new_target.sh* del servidor de almacenamiento y *new_instance.sh* del servidor de virtualización.

Limpieza de usuarios (cleanup.sh)

Este programa se limita a ejecutar los scripts *cleanup.sh* de los servidores de almacenamiento y virtualización.

Sitio web

El sitio web actualmente consta de un portal de inicio de sesión y un pequeño *script* que ejecuta la redirección al endpoint del servidor de virtualización si obtiene un *login* correcto.

El portal de *login* es una modificación del *template Colorlib Login Form V16* [7].

El código que maneja el inicio de sesión [17] está escrito en PHP, contiene la lógica vista a continuación:

```
1 Obtenemos las strings asociadas al usuario y la contraseña del
      formulario
2 Procesamos las strings para eliminar posibles inyecciones SQL
3 Si el usuario o la contraseña están vacíos
4     Mostramos un error y terminamos
5 Ejecutamos la consulta que cuenta el número de entradas
      correspondientes al usuario y contraseña dados en la BBDD
6 Si cuenta <= 0
7     Mostramos un error y terminamos
8 Efectuamos una redirección al puerto del endpoint del servidor de
      virtualización pasándole el nombre de usuario
```

6.3 Sofware de la instancia Clobber

La propia instancia se comunica con el exterior a través de un servicio expuesto en el puerto TCP *5000*. Haciendo uso de las *APIs* ofrecidas por este servicio podemos tanto añadir comandos nuevos a la librería como ejecutar órdenes ya cargadas.

6.3.1 Dockerfile

El Dockerfile es el fichero que contiene las órdenes necesarias para generar una imagen Docker. En este caso contiene las siguientes instrucciones:

```
1  Parte de la imagen base de Ubuntu
2  Instala python y ssh
3  Crea la carpeta /server y copia en ella el código del endpoint
4  Instala las librerías necesarias
5  Cambia la contraseña del superusuario a clobber
6  Habilita el inicio de sesion de root por ssh
7  Añade el usuario clobber con contraseña clobber
8  Crea la carpeta /scripts
9  Ejecuta el servicio ssh
10 Crea las carpetas de comandos y datos de usuario
11 Lanza el servicio en el puerto 5000
```

6.3.2 Parser clobber

El parser de cada instancia Clobber contiene las *APIs* necesarias para la configuración y puesta en marcha de la herramienta:

- Comprobar el estado del servicio.
- Acceder a una aplicación.
- Ejecutar un comando.
- Añadir un comando.
- Eliminar un comando.

Comprobar el estado del servicio

Este endpoint del API nos devuelve un JSON con el valor *OK* en caso de que el servicio esté activo y sea accesible.

Acceder a una aplicación

Este endpoint toma un parámetro desde la URL y trata de redirigirnos al *template* asociado.

Ejecutar un comando

Este endpoint envía una orden al motor de ejecución de Clobber para que ejecute la orden especificada. Podemos aportar parámetros al comando separados por slashes `/`. La acción a ejecutar debe haber sido registrada previamente.

Añadir un comando

Los comandos en clobber se definen mediante *reglas*. Las reglas se componen de los siguientes archivos:

- Definición de regla: *rule.json* **obligatorio**
- Script con el comando a ejecutar: *script.sh* opcional
- Script con el filtro para *stdout*: *filter.sh* opcional

Estos deben comprimirse en bajo el nombre del comando a registrar con la extensión .zip.

A su vez, el archivo para la definición de comandos se compone de los siguientes apartados:

- **name**: Nombre del comando, debe coincidir con el del fichero comprimido, **obligatorio**.
- **command**: Comando a ejecutar, ignorado si *script.sh* está presente, **obligatorio**.
- **filter**: Filtro para stdout, ignorado si *filter.sh* está presente, opcional.
- **errfilter**: Filtro para stderr, opcional.
- **async**: Flag que determina si el comando bloqueará la ejecución. *Actualmente sin implementar.*

Este endpoint contiene un formulario para cargar nuestro comando en el caso de ser invocado por una petición de tipo GET, este formulario llama a su mismo endpoint empleando una petición POST para entregarle los datos que debe descomprmir y registrar.

Eliminar un comando

Actualmente, este endpoint elimina todos los comandos vaciando la caché y eliminando la carpeta con los comandos registrados.

6.3.3 Motor de comandos

El motor es el módulo encargado de ejecutar un comando registrado. Para ello lanza un proceso independiente al que previamente ha secuestrado los descriptores *0,1 y 2* (*stdin*, *stdout y stderr* respectivamente). Una vez el comando inicial haya finalizado, aplicará (si los hubiera) los filtros asociados a cada salida.

Por último, con el propósito de respetar el Principio de Responsabilidad Única, designamos al motor como encargado de registrar un nuevo comando a partir del JSON de definición.

6.3.4 Caché de comandos

Los comandos una vez registrados se serializan mediante la librería SqliteDict. A pesar de que esta tiene un rendimiento muy alto, incluimos una caché de comandos usados recientemente.

Para implementala haremos que todas las lecturas a la BBDD pasen previamente por un módulo (*DBManager.py*) cuyo comportamiento es el siguiente:

```
1 funcion obtenerComando(nombre)
2     Si comando no en la caché
3         Traer comando a la caché
4     Devolver comando de la caché
```

6.3.5 Librería Javascript

Para interactuar con Clobber desde los diferentes templates, proporcionamos un breve fichero *Javascript* que permite ejecutar un comando mediante JQuery de manera sencilla.

Este código genera la URL de la petición a partir del nombre del comando y un elemento HTML que contenga en su parámetro val los argumentos para la petición y posteriormente, ejecuta la orden $.getJSON con el callback proporcionado por el usuario.

Capítulo 7

Ejemplo de implementación

HASTA ahora hemos descrito los servicios, la interacción entre servidores y el software que actúa en cada máquina. Sin embargo, aún no hemos definido el modo de interactuar con la solución. *Recordemos que el usuario de Clobber es un programador elaborando su propio servicio.*

Durante este capítulo explicaremos como llevaríamos a cabo el desarrollo de un pequeño programa de edición de textos en línea, que leyera y guardara los ficheros desde la máquina del consumidor.

7.1 Definición del API

Clobber sigue la metodología API-FIRST. Según esta metodología, el primer paso al comienzo de un proyecto es plasmar toda su interacción en la definición de un API que será empleada tanto como especificación de funcionalidades como parte de los requisitos de aceptación y *pacto* con el cliente. Para determinar que funcionalidades debe incluir este API, primero debemos enumerar los casos de uso de nuestro programa:

En este caso, nos limitamos a **leer** y **escribir** archivos de texto.

Para cada una de estas funciones podemos documentar un endpoint de nuestra API haciendo uso de herramientas como Swagger.

Una definición para estos endpoints en *pseudocódigo* sería:

```
1 cargaFichero:
2     recibe: Nombre del fichero
3     devuelve: Contenido del fichero
4 grabaFichero:
5     recibe: Nombre del fichero
6     devuelve: Código de error
```

La definición completa, visible en la Figura 7.1, se puede consultar en GitHub

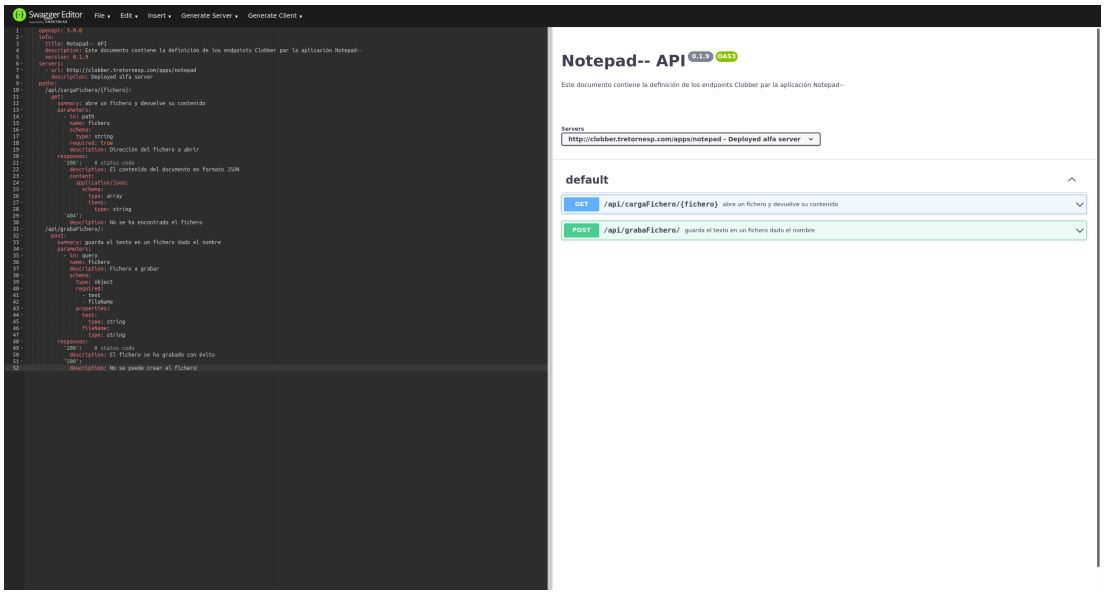

Figura 7.1: Endpoint documentado en *Swagger*

7.2 Creación de los comandos

Para cada uno de estos endpoints, debemos crear un fichero de definición de comandos. Siendo esto una prueba de concepto, es suficiente ejecutar cat sobre el primer parámetro para el endpoint *cargaFichero* y un echo con stdout redirigido para el endpoint *grabaFichero*. Al no necesitar filtros, nuestro *rule.json* se parecería a:

```
\{
  "name": "cargaFichero",
  "command": "cat \"*%params%*\""
\}
```

El elemento *params* hace referencia a los parámetros del endpoint. En este caso el nombre de fichero a cargar. Finalmente subimos los comandos creados mediante el formulario de registro de comandos.

7.3 Programación del frontend

Llegado a este punto, solamente necesitamos programar la interfaz de nuestro editor de textos mediante la tecnología web que más nos guste. Siguiendo con el ejemplo, diseñaríamos un sitio que contuviera:

- Un cuadro de texto editable.
- Un botón asociado a la acción de abrir fichero.
- Un botón asociado a la acción de guardar fichero.

El resultado podría verse como en la Figura 7.2

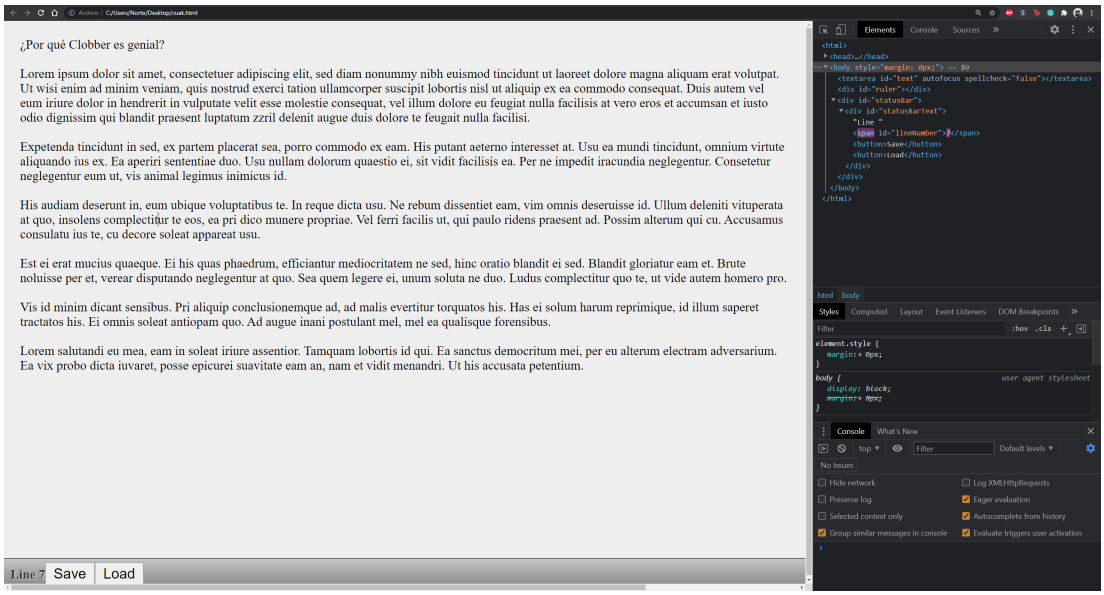

Figura 7.2: Posible vista de la aplicación

7.4 Integración de Clobber

Para integrar clobber en nuestra aplicación web tenemos que seguir los siguientes pasos:

1. Incluir JQuery.
2. Incluir la siguiente línea en el <*head*> de nuestra aplicación:

```
<script type=text/javascript>$SCRIPT_ROOT = {{ request.script_root|
    tojson|safe }};</script>
```

3. Incluir la pequeña librería *clobber.js* (Sección 6.3.5) en el <*head*> de nuestra aplicación.

7.5 Conexión con los endpoints

Finalmente, podemos asociar el evento que queramos a la función *clobber*, cuya sintaxis es la siguiente:

clobber(*comando,argumento,callback*)

- *Comando*: *url* del *endpoint* registrado.
- *Argumento*: elemento *html* que contiene el argumento. Normalmente una caja de texto.
- *Callback*: función que se ejecutará una vez la petición retorne.

Siguiendo este ejemplo, podremos añadir dos llamadas a la función *clobber*: una para **leer** y otra para **escribir**, invocando a los respectivos endpoints, pasándoles como argumento una caja de texto que contuviera el texto a leer/escribir por un lado y el nombre del fichero de origen/destino por otro.

7.6 Inclusión de la app en el contenedor

Finalmente solo necesitamos subir la aplicación a la carpeta templates de la instancia clobber. Es importante mencionar que estos templates son renderizados por Flask, que emplea el parser Jinja2, por lo que debemos respetar la estructura de carpetas y formato específicos para la inclusión de ficheros de recursos.

Capítulo 8

Rendimiento

En este capítulo hablaremos de como hemos obtenido las métricas de rendimiento, mostraremos los resultados obtenidos a través de tablas comparativas y finalmente enunciaremos nuestras conclusiones. También haremos mención a las limitaciones del entorno de pruebas.

8.1 Selección de métricas

Los servidores que soportan la carga de Clobber son *virtualización* y *almacenamiento*. Su demanda de recursos crecerá conforme se vayan generando instancias. Centraremos por ello el grueso de nuestras pruebas en estas dos máquinas.

En el caso del servidor de almacenamiento las cuestiones a responder fueron:

- Número de targets iSCSI que podemos generar como máximo.
- Curva de utilización de CPU y memoria principal frente al número de targets.
- Tiempo medio de creación de un target frente al número de targets.

En relación al primer ítem, a pesar de que la documentación indica que este número se corresponde con *1000* targets, nos gustaría comprobarlo empíricamente.

Para el servidor de virtualización determinamos:

- Número de instancias que podemos generar como máximo.
- Tiempo medio de creación de una instancia frente al número de instancias.
- Curva de utilización de CPU y memoria principal frente al número de instancias.

Estas métricas nos aportan cierto conocimiento de la **capacidad** bruta del sistema. Para analizar su escalabilidad, repetiremos el análisis para las siguientes configuraciones:

- 1 núcleo, 4GB de RAM.
- 1 núcleo, 8GB de RAM.
- 4 núcleos, 8GB de RAM.
- 8 núcleos, 8GB de RAM.
- 4 núcleos, 12GB de RAM.

Por último determinaremos el tiempo de respuesta de una petición HTTP sobre el servicio.

8.2 Software de pruebas

El programa que empleamos para inducir al sistema la carga necesaria es denominado *STRESS.sh*[1]. Este script es capaz de inciar de manera automática el proceso de registro de una nueva instancia con su almacenamiento asociado.

STRESS recibirá por parámetro el *número de instancias a registrar*, el *espacio de almacenamiento para cada instancia*, el *tiempo de espera máximo para una petición HTTP* y el *directorio donde guardar las métricas*.

Una vez invocado, seguirá los siguientes pasos:

1. Trata de levantar el número especificado de targets iSCSI.
2. Guarda en formato CSV los tiempos asociados a levantar cada target iSCSI.
3. Trata de levantar el número especificado de instancias.
4. Guarda en formato CSV los tiempos asociados a levantar cada instancia.
5. Ejecuta una petición al endpoint de testing de cada instancia
6. Comprueba si la instancia es accesible, el *endpoint* existe y el almacenamiento remoto está presente.
7. En caso negativo, guarda información relevante en un fichero de log.
8. Nos muestra por pantalla el resumen del tiempo que tomó en cada apartado.
9. Llegado a este punto podemos optar por repetir el paso 5 o limpiar los datos de prueba.

Las salidas stdout y stderr de los pasos *1* y *3* nos permiten observar si se ha producido algún error en el proceso de creación. Finalmente comparamos la petición del paso *5* para determinar si la instancia se ha levantado correctamente.

[1] Disponible en https://github.com/TretornESP/Clobber/blob/main/webserver_scripts/STRESS.sh

Determinar el estado de la instancia

Para poder determinar si una instancia se ha inicializado correctamente, hemos modificado el endpoint de pruebas. Su nuevo comportamiento es el siguiente: una vez ejecutada la petición, el servidor tratará de abrir la carpeta de datos (alojada en el servidor de almacenamiento) y devolverá como respuesta el sistema de ficheros que haya detectado en ella. Comparando esta respuesta del endpoint con la cadena de caracteres **ext4**, determinamos si una instancia ha sido desplegada y si tiene acceso al almacenamiento compartido.

8.3 Especificación del entorno de pruebas

El equipo anfitrión cuenta con las siguientes especificaciones técnicas:

- Procesador AMD Ryzen 3800X @ 3.9GHz, 8 núcleos (16 hilos).
- Memoria RAM: 16GB (2x8GB) Kingston DDR4 a 3200Mhz PC-25600 CL16.
- Almacenamiento 500GB Samsung SSD 980 NVME M.2.
- GPU AMD RX 5700XT 8GB GDDR6.
- Placa MSI Gaming Plus MAX (PCIE 3.0).

El sistema operativo anfitrión en reposo dispone de *13600MB* de memoria principal libres y hace un uso de menos del *<1%* de la CPU.

El entorno ejecuta *4* máquinas virtuales para *router*, *virtualización*, *web* y *almacenamiento*. La ejecución concurrente de estas instancias resulta en un consumo de *7100MB* de memoria aproximadamente. Por tanto nuestras pruebas podrán ocupar unos *6500MB* antes de incrementar el uso del swap.

8.4 Resultados de las pruebas

Tras la ejecución de la heramienta *STRESS* hemos obtenido los resultados recogidos en la Tabla 8.1. Los tiempos mostrados son la media a **5** ejecuciones con un valor de almacenamiento de **100MB** (usamos caché mediante rsync).

Por último se realizaron varias pruebas de ruptura con *250* y *1000* instancias, de las que se extraen los siguientes resultados:

- El servidor de almacenamiento logró desplegar **931** targets como máximo antes de comenzar a fallar.

	Nº	Tiempo Almacenamiento	Tiempo Virtualización	Tiempo Respuesta	Errores
1 núcleo 4GB	10	400	1200	300	0
	25	480	1320	360	0
	50	520	1520	360	0
	100	530	2170	420	0
1 núcleo 8GB	10	600	1100	400	0
	25	520	1280	320	0
	50	520	1520	340	0
	100	530	2080	360	0
4 núcleos 8GB	10	500	700	200	0
	25	520	720	280	0
	50	500	740	260	0
	100	500	830	270	0
8 núcleos 8GB	10	500	700	300	0
	25	480	720	280	0
	50	500	760	280	0
	100	500	840	290	0
4 núcleos 12GB	10	500	700	300	0
	25	520	720	280	0
	50	540	760	280	0
	100	530	880	280	0
	150	540	966	293	0

Tabla 8.1: Resultados de las pruebas de carga (tiempos en ms)

- El servidor de virtualización comenzó a hacer uso extensivo del swap al alcanzar las **145** instancias, abortó la ejecución en **175**.

8.5 Interpretación de resultados

En la Figura 8.1 podemos observar la aproximación de los tiempos de ejecución correspondientes a **150 instancias** en la configuración **4 núcleos 12GB de memoria**. Los elementos en *verde* se corresponden a los tiempos de ejecución del servidor de virtualización, mientras que los elementos en *azul* se corresponden a los tiempos del servidor de almacenamiento. El *eje 'x'* indica el número de instancias activas en el rango *0-150* y el *eje 'y'* indica el tiempo en *segundos* dentro del rango *0.1-1.3*.

- Se puede observar una disminución del tiempo de ejecución al pasar de *1* a *4* núcleos. Esto se debe a que en el primer caso, se producía un cuello de botella, pasando de *4* a *8* núcleos no percibimos ninguna mejora significativa, con lo que podemos afirmar que

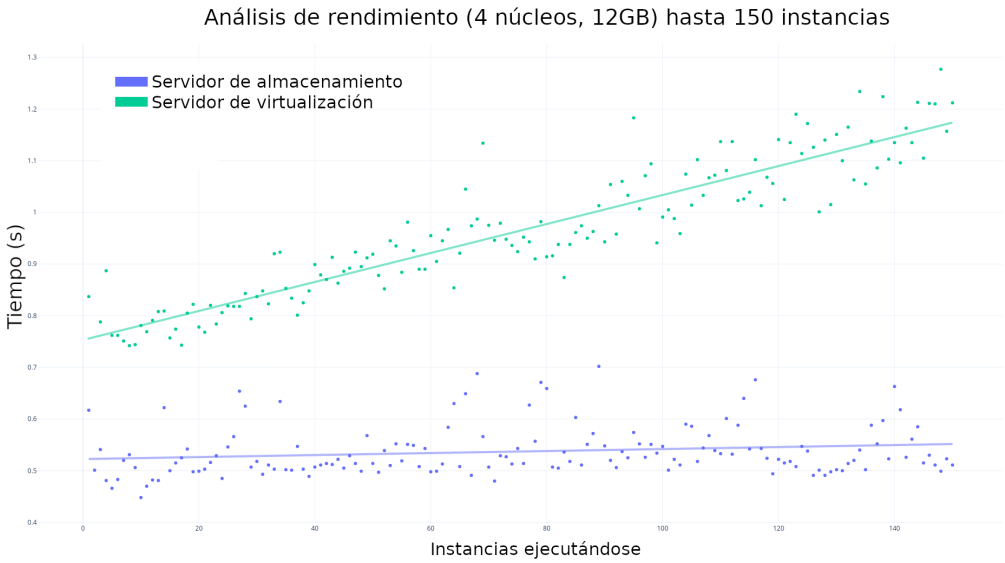

Figura 8.1: Tiempo de ejecución en función del número de instancias

este sistema **no escala con respecto al número de núcleos.**

- Podemos observar que la memoria disponible tiene un impacto pequeño en el tiempo de ejecución, el principal factor que determina es el *número máximo de instancias* que podemos tener activas de forma concurrente. Del análisis de uso de memoria y las pruebas de ruptura obtenemos la cifra de **37MB** de espacio consumido por cada instancia levantada. Teniendo en cuenta que es recomendable mantener una cuarta parte de la memoria física libre para incrementar la estabilidad del servidor y minimizar el uso del swap, podemos calcular la memoria necesaria para alojar un número determinado de instancias dividiendo la memoria disponible entre el tamaño de cada instancia. A esta cifra hay que restarle la memoria que consumen el sistema operativo y las demaś utilidades del anfitrión. En el caso del servidor de virtualización, esta cantidad es de *1024MB* con lo que la fórmula resultante sería (Ecuacion: 8.1).

$$f(n) = (3700 * n + 102400)/75 \ = 50n + 1365 \tag{8.1}$$

Despejando la n de la ecuación podemos determinar cuantas instancias podemos ejecutar como máximo en nuestro equipo:

$$t(m) = (m - 1365)/50$$

Introduciendo la memoria de la que disponemos (*6500MB*) obtenemos una cifra de **102**

instancias. Exceder este número podría implicar el uso del swap, con lo que los tiempos de ejecución bajarían considerablemente, dificultando su comparación las métricas de otros escenarios. Por este motivo, en ausencia de información empírica, deberíamos emplear para nuestra recta de ajuste los datos para *100 iteraciones*. Sin embargo, se determinó experimentalmente que el sistema actuaba de forma lineal hasta las **150** instancias, con lo que podemos ampliar nuestra muesta hasta ese rango.

- El tiempo de respuesta se mantiene entre los **300 y 400 ms** sin mucha variación. Este tiempo resulta aún **demasiado alto** para el despliegue de un servicio comercial. En el futuro se debe realizar un análisis más exhaustivo con el fin de determinar la causa de esta latencia.
- El servidor de almacenamiento soportó **931** targets antes de fallar, posteriormente se descubrió que el número de dispositivos propios del sistema aproxima al medio centenar. Estos resultados se alinean con la documentación que afirmaba que existía un límite de *1000* dispositivos presentes de manera simultanea.
- El servidor de virtualización comenzó a hacer uso del swap a las **145** instancias, resultado que se asemeja mucho al resultado de nuestra función sin margen de seguridad $6500 - 1024)/37 = 148$.
- En total se levantaron $5 \times 150 + 5 \times 5 \times (100 + 50 + 25 + 10) = 750 + 25 \times (185) =$ **5375** instancias, de las cuales **5374** pasaron la prueba y **1** falló. *Se sospecha que por falta de memoria disponible.*

Finalmente damos respuesta a las cuestiones iniciales:

- El número de máximo de targets debe ser inferior a **1000**.
- El uso de CPU del servidor de almacenamiento **no varía** en función del número de targets.
- El tiempo medio requerido para la creación de un target **no varía** en función del número de targets.
- El número de instancias que podemos generar, con seguridad, como máximo viene definido por la función: $r(m) = (m - 1365)/50$ donde *m* es la memoria disponible en MB.
- El tiempo de despliegue de una instancia con relación al número de instancias se define como $s(n) = M * n + B$ siendo $M = 0.0028$ y $B = 0.7531$ donde *n* es el número de instancias ejecutándose.

- La curva de utilización de CPU se corresponde con una **constante**. La curva de uso de memoria viene dada por la Ecuación 8.1
- El tiempo de respuesta del servicio en nuestro entorno de pruebas es de unos **350ms**.

Capítulo 9

Conclusiones y trabajo futuro

Clobber representa una prueba de concepto. El potencial de esta solución no obtendrá pleno alcance hasta que se extienda en múltiples direcciones. El objetivo principal de este capítulo es ilustrar las líneas de trabajo necesarias antes de su lanzamiento.

9.1 Conclusiones

El producto conseguido al final de este proyecto cumple con los objetivos expuestos en la Sección 1.3 de este documento. Hemos hecho un estudio en profundidad de las tecnologías necesarias para un buen desarrollo de este tipo de sistemas. Con este trabajo hemos conseguido un sistema que permite dotar a los programadores de un entorno potente, flexible y dedicado sobre el que ejecutar sus programas. Exponemos ahora la consecución de nuestros objetivos asimilando a cada uno, software dentro de nuestro repositorio[1] o, en su defecto, indicando qué módulo se hace cargo.

1. Soportar un API capaz de describir la interacción entre el sistema operativo y la aplicación: este punto se materializa en el *módulo de registro de APIs* y el esquema de reglas (fichero command-schema.json).

2. Diseñar e implementar un parser capaz de generar, a partir de un fichero, aplicaciones que interactúen con Clobber: el código encargado de esto se puede consultar en el fichero main.py.

3. Diseñar e implementar un gestor designado para labores de *autenticación, manejo de servidores y peticiones HTTP*: de estas tareas se encarga el *Servidor web* a través de los scripts contenidos en el directorio webserver_scripts.

[1] https://github.com/tretornesp/clobber

4. Elaborar un escritorio virtual a modo de demostración: el código frontend del ejemplo se puede consultar en el fichero desktop.html y su fichero de reglas en el fichero rule.json.

5. Programar un módulo de comandos capaz de ejecutar órdenes arbitrarias y devolver su resultado en un formato estandarizado: de esto se encarga el módulo *commander*, implementado en command.py.

6. Realizar un plan de despliegue para la infraestructura: el documento técnico de infraestructuras, presente en el fichero tfg.md del repositorio, define este proceso.

Como autor, he ganado mucha experiencia en administración de sistemas *Linux*, programación de *Shell Scripts* ¡Al fin fui capaz de usar "SED" y "AWK" sin buscar tutoriales! He estudiado el funcionamiento interno de Docker y ampliado conocimiento sobre redes de almacenamiento y el protocolo iSCSI. Finalmente, gané soltura con LaTeXe hice una primera toma de contacto con el mundo académico a través de la elaboración de esta memoria.

9.2 Líneas de trabajo futuras

Con respecto al contenido de este trabajo, entendiéndolo como un subconjunto de la solución completa que es Clobber, valoramos una serie de funcionalidades que no han sido incluidas en esta versión. Son las siguientes, en orden descendente de importancia:

9.2.1 Mejoras en el servicio de almacenamiento

Actualmente el servicio de almacenamiento solo nos permite crear discos ext4 thick-provisioned, por lo que las siguientes propuestas serían deseables:

Cifrado en dos etapas de los discos

En esta mejora, emplearíamos la clave del usuario para cifrar el disco cuando este estuviese inactivo. El nivel de seguridad variaría en función de si el usuario estuviera conectado, inactivo o desconectado. La principal que presenta esta funcionalidad problemática es el nivel de optimización requerido para asegurarnos unos tiempos de disponibilidad razonables independientemente del tamaño del almacenamiento.

Thin-Provisioning

Actualmente el tamaño de los discos se especifica en *megabytes* cuando iniciamos una nueva instancia. Inicialmente este disco contendrá un sistema de archivos prácticamente vacío, lo que malgasta una gran cantidad de espacio. La solución del thin-provisioning pasa por

incrementar o disminuir el tamaño del disco en función de la demanda que requiera el usuario. La principal problemática parte de la investigación necesaria para redimensionar el disco sin alterar la consistencia del sistema de ficheros. Proponemos una solución *sub-optima* que consiste en generar a intervalos regulares un nuevo fichero de disco, del tamaño correspondiente a la región ocupada del disco a redimensionar, copiar todos los ficheros a este nuevo dispositivo y borrar el anterior. Debido a la probable lentitud de este mecanismo, optamos por mover esta característica a la lista de trabajo futuro e investigar otras vías de acción.

Modificar el sistema de ficheros

Estudiamos la posibilidad de especificar el sistema de ficheros con el que trabajar ya que actualmente solo estamos actuando con ext4. La inclusión de esta funcionalidad es casi trivial (modificar un parámetro en el comando mkfs) sin embargo la falta de aplicación práctica en el estadio actual nos lleva a posponerlo.

9.2.2 Mejoras en la API de administración de la instancia

En el caso de pretender que la aceptación de Clobber sea alta, necesitamos que la programación con la herramienta sea lo más sencilla posible. Para facilitar este proceso debemos expandir el API proporcionada. Estas son algunas propuestas:

Agregación de páginas a través de un endpoint

Los templates se podrían añadir de la misma manera que añadimos actualmente los comandos; para esto sería necesario desarrollar un clasificador que situara los ficheros en las carpetas adecuadas de acuerdo a la especificación de Flask. Debido a que añadir los templates de manera manual resulta ya sencillo hemos decidido posponer esta tarea.

Generar los comandos desde una definición OpenAPI

Este apartado sería tal vez el más importante desde el punto de vista del usuario. Generaremos los ficheros de reglas *rules.json* a partir de una definición estandarizada de los endpoints. Esto facilitaría en gran medida la aplicación de la metodología API-FIRST y resultaría un punto de casi interés publicitario para el software. La principal problemática consiste en el amplio repertorio de casos límite que deberíamos comprobar como seguro de una traducción correcta.

9.2.3 Mejoras de seguridad

Que en esta versión la seguridad no es prioritaria es un secreto a voces. Un atacante con acceso al código fuente tardaría pocos minutos en obtener acceso a los servidores de alma-

cenamiento. Esto, sumado a la ausencia de cifrado, supondría el acceso a los datos de todos los usuarios. A continuación exponemos diferentes ideas para incrementar la seguridad del sistema.

Autenticación avanzada a nivel de instancia

Actualmente es el servidor de virtualización quien se sitúa como punto intermedio entre los consumidores y las instancias. Sería una posibilidad real la protección del puerto *5000* interno de cada instancia, tal vez mediante filtrado de paquetes. Por último, resulta imperativo incluir algún sistema de control de sesiones vinculado a los endpoints de administración de la instancia.

Cifrado en las comunicaciones

El uso de comunicaciones cifradas es una necesidad en el caso de salir a un entorno de producción. No hemos aplicado esta idea hasta ahora, debido a que dificulta el análisis del tráfico en fase de pruebas.

9.2.4 Mejoras en la infraestructura

La arquitectura propuesta contempla un cierto grado de escalabilidad y Tolerancia a fallos aunque, actualmente, no hemos alcanzado el proceso de implementación. El motivo es que pretendemos realizar un eventual despliegue empleando servicios de computación en la nube (probablemente AWS). Debido a la inherente dependencia de las técnicas a aplicar con la infraestructura, las funciones que incluiríamos en esta categoría, tales como *balanceo de carga y creación / destrucción de servidores bajo demanda*, no resultan factibles sin contar con la plataforma subyacente. Debo puntualizar que realizamos el despliegue del entorno de pruebas en un equipo físico debido a lógicas restricciones económicas.

9.2.5 Lineas de trabajo externas

Fuera del marco estricto de este proyecto se encuentran varias propuestas necesarias para completar Clobber en toda su extensión:

Dispositivo hardware

El máximo potencial de Clobber se alcanza en un dispositivo móvil. Un diseño propio para este equipo aportaría ventajas tales como *mayor duración de batería, menor peso y bajo coste.*

Se presupone que este dispositivo debería contar con las siguientes caracterísiticas:

- Procesador ARM.

- Capacidad para conectividad móvil mediante 5G y WIFI.
- Tarjeta de red dedicada exclusivamente a notificaciones push.
- Unidad de compresión de vídeo por hardware.

Una de las cualidades más interesantes de este dispositivo es que en principio no almacena estado. Así la acción de *bloquear* –tradicional en un teléfono ó tablet–, podría equipararse al estado HALTED de una *CPU* sin mayor dificultad. Aquí entra en juego la *tarjeta de red dedicada*: esperando la recepción de un paquete específico, a cuya llegada interrumpe a la *CPU*, que procesa la orden y vuelve a su estado de espera.

Sistema operativo

El sistema operativo que ejecutaría este hardware dedicado se liberará de cierta carga asociada al los permisos de usuario, manejo de ficheros (no necesitamos almacenar mas que ficheros temporales y de caché) y concurrencia, ya que en principio el número de procesos sería predecible. Se encontrará en un punto medio entre un firmware y un kernel completo, con la particularidad de que el desarrollo de *device drivers* debería seguir un esquema robusto, ya que debe portarse a multitud de dispositivos muy heterogéneos.

Programa principal

Tal vez este punto nunca llegue a ser necesario. Es cierto que el concepto de Launcher está muy extendido y nos resulta familiar a todos. Sin embargo, en el caso de las computadoras de escritorio y en base a mi experiencia personal, la proporción de tiempo haciendo uso del navegador supera ampliamente al resto de programas. Tal vez en Clobber, la página de *Google* sea el único launcher que necesitemos.

De todas formas, reconocemos la necesidad de un programa principal para quien haga uso de Clobber en dispositivos de propósito específico como *sistemas de control, domótica y automatización*, lo que es perfectamente factible. Este desarrollo queda fuera de nuestras competencias.

9.3 Relación con la titulación

Durante el transcurso de este trabajo, hemos puesto en práctica multitud de habilidades adquiridas a lo largo del grado. En concreto, el despliegue y configuración de la infraestructura se fundamentaron en los conocimientos adquiridos durante las asignaturas de *Administración de Infraestructuras Informáticas* y *Redes*. La estructura en pequeños programas que interactúan

mediante *endpoints* hereda de las arquitecturas RESTful vistas en *Internet y Sistemas Distribuidos*. El uso de redes de almacenamiento SAN mediante el protocolo iSCSI fue estudiado en la asignatura *Administración de Infraestructuras Informáticas*; las etapas de planificación y elaboración del proyecto fueron tomadas de la formación en la herramienta MS Project de la asignatura *Gestión de Proyectos*. Por último, el uso de las tuberías para redirección de *descriptores de ficheros*, así como el conocimiento abstracto pero necesario en el funcionamiento del *kernel Linux* viene dado por la asignatura de *Sistemas Operativos*.

Éstos son algunos ejemplos de asignaturas que han influido directamente en mi capacitación a la hora de llevar a cabo este proyecto; sin embargo, no son todos: *la mentalidad del ingeniero, a pesar de ser un concepto abstracto, es lo que más me ha ayudado a completar esta propuesta.*

9.4 Observaciones finales

La idea que acompaña a Clobber llevaba rondando mi cabeza ya un par de años, aunque nunca pensé que eventualmente se convertiría en el trabajo con el que finalizo mi etapa universitaria. Desde un primer momento, el objetivo había sido elaborar un software abierto, muy vistoso y con una base de código fácil de ampliar, para que un día, con suerte, un alto ejecutivo de alguna corporación lo viera y le inquietase. No por Clobber ni por mi, nada más lejos de la realidad, si no por la posibilidad, aunque remota, de que algún competidor suyo decidiera poner en marcha el proyecto con los recursos necesarios; porque la arquitectura es tan sencilla que se puede llevar a cabo en un par de meses por un estudiante de cuarto de carrera, porque cumple su propósito con cierta soltura y porque es retrocompatible con equipos antiguos. Este último punto es muy importante ya que, de requerir hardware nuevo sería posible que antepusieran los beneficios económicos a corto plazo justificando así la implementación. Al poder emplear hardware actual –cualquier dispositivo con un navegador– no existe, en mi opinión, motivación alguna por parte de un gran fabricante de aplicar la plataforma. No obstante, y por su propia naturaleza, resulta muy factible que un agente disruptor decidiera lanzar algún producto combinando esta tecnología con un modelo de monetización basada en servicios. Eso, con mucha suerte, iniciaría una escalada de competencia, propiciada por el voraz sistema capitalista que inevitablemente desembocaría en el fin de la obsolescencia programada en los dispositivos móviles.

> *The name's Lanley. Lyle Lanley. And I come before you good people tonight with an idea. Probably the greatest... Aw, it's not for you. It's more of a Shelbyville idea.*
>
> *Lyle Lanley*

Lista de acrónimos

AJAX Asynchronous Javascript And XML. 6

API Application Programming Interface. 3, 6, 20, 45, 49, 61, 63

ARM Advanced RISC Machine. 64

AWS Amazon Web Services. 64

BBDD Bases de Datos. 46

CSS Cascading Style Sheets. 11, 21

CSV Comma Separated Value. 54

CU Caso de Uso. 26

DD Dataset Definition. 14

DHCP Dynamic Host Configuration Protocol. 3, 12, 33, 34

DNS Domain Name Server. 3, 12, 33, 38

ECTS European Credit Transfer System. 19

GIMP GNU Image Manipulation Program. 15

GPLv3 GNU Public License Version 3. 1

GPS Global Positioning System. 2

HH Horas-Hombre. 18, 19

HTML HyperText Markup Language. 11, 47

HTTP HyperText Transfer Protocol. 4, 6, 12, 30, 54, 61

IBM Integrated Business Machines. 7

IOCTL Input/Output Control. 2, 31

IP Internet Protocol. 6, 12, 18, 27, 30, 35, 38, 40, 41

IPv4 Internet Protocol Version 4. 34

IQN iSCSI Qualified Name. 38, 39

iSCSI Internet SCSI. 3, 6, 13, 14, 20, 27, 30, 37, 38, 53, 54, 62, 66

JSON JavaScript Object Notation. 18, 28, 45, 46

LAMP Linux Apache Mysql and PHP. 12, 30

LUN Logical UNit. 38, 39

LwIP LightWeight Internet Protocol. 31

MAC Medium Access Control. 34, 35

NAT Network Address Translation. 41

PHP Personal Home Page. 11–13, 21, 30, 35, 36, 44

PID Process IDentifier. 6

PXE Preboot eXecution Environment. 8

RAM Random Access Memory. 54

SAN Storage Area Network. 66

SCO Santa Cruz Operation. 10, 80

SCSI Small Computer System Interface. 6

SED Stream EDitor. 62

SSH Secure SHell. 13, 30, 31, 35, 38, 42, 43

TCP Tansmission Control Protocol. 44

URL Uniform Resource Locator. 6, 43, 45, 47

WAN Wide Area Network. 9

XML eXtended Markup Language. 6

Glosario

$.getJSON Orden JQuery que permite solicitar un JSON a un endpoint de manera asincrona especificando una función como callback. 47

10Zig Cliente thin propiedad de 10ZiG. 9

0 El descriptor 0 (stdin) está asociado a al buffer de entrada de la terminal de la que depende el proceso actual. 5, 46, 79

1 El descriptor 1 (stdout) se asocia a la salida estandar de la terminal del proceso. 5, 18, 46, 79

2 El descriptor 2 (stderr) se asocia a la salida dedicada a errores de la terminal del proceso. 5, 18, 46, 79

acelerómetro Sensor capaz de medir aceleración. 2

actor En este contexto hace referencia a quien incia una secuencia de acciones. 25, 26

adaptador puente En VirtualBox, simula una tarjeta de red física conectada diréctamente al router de la máquina anfitrión. 34

alfa Versión de evaluación que no incluye todas las funciones del producto final. 18

anfitrión En terminos de virtualización, un anfitrión es la máquina, normalmente física, que ejecuta una máquina virtual. 1, 5, 18, 29, 55, 57

Apache Servidor HTTP propiedad de la Apache Foundation. 12, 30, 35, 36

API-FIRST Metodología de desarrollo que toma el API como pilar principal del proceso de fabricación. 49, 63

arquitectura de microservicios Enfoque de desarrollo basado en dividir un proyecto en múltitud de programas de tamaño reducido y que interactúan entre si y con el usuario. 1

ASTER Software de Multiseating para Windows propiedad de Ibik. 7

Atom Software de edición de textos propiedad de GitHub Inc. 14

AWK Herramienta Linux que permite buscar y procesar patrones. 62

balanceo de carga Contando con un servicio replicado, redirigir las peticiones entrantes hacia el nodo mas adecuado (ya sea por localización geográfica, porcentaje de carga del nodo, etc). 64

Bind9 Software de servidor DNS propiedad del Internet Systems Consortium. 12, 35

bug Error informático. 7

caché Espacio de almacenamiento comparativamente mas rápido y pequeño que otro y al que se superpone. Nos permite aplicar el principio de localidad para aumentar las prestaciones. 3, 12, 33, 46, 65

caja negra Que no tiene conocimiento del funcionamiento interno del sistema, lo entiende como una función que transorma entradas en salidas. 27

callback Función que se ejecuta a raíz de un evento. 6, 47

cascada Metodología clásica de desarrollo que sigue las fases *análisis, diseño, implementación y pruebas* de manera secuencial. 17

cat Comando Linux que redirige la entrada stdin a la salida stdout. 50

certificados Un certificado digital permite identificar tanto a su portador como al software que haya firmado. 38

CGroup Grupo de control. Agrupación de procesos destinada a administrar los recursos que consumen de manera conjunta. 6

Citrix Conjunto de soluciones de virtualización, computación en la nube y acceso remoto propiedad de la compañía Citrix Systems. 9

clave primaria En SQL, hace referencia al campo único capaz de identificar unívocamente las diferentes entradas de una tabla. 30

Clearcube Cliente thin propiedad de la empresa ClearCube. 9

Clonezilla Software libre para la configuración de servidores de clonación propiedad del NCHC Free Software Labs. 8

complejidad ciclomática Medida de la complejidad lógica de un algoritmo. 20

concurrencia Ejecución simultánea aparente de dos o mas programas. La aparencia de paralelismo se logra haciendolos entrar y salir de la CPU rápidamente, mediante funciones no reentrantes *yield* e interrupciones *preempting*. 5, 65

contenedor En este contexto: máquina virtual Docker independientemente de su estado. 27, 30, 41

cortafuegos Software que se sitúa en medio de dos equipos, interceptándo todas sus comunicaciones y llevando a cabo tareas de filtrado con el fin de aumentar la seguridad. 3, 74

cuello de botella Ralentización debida a un componente que opera a una velocidad inferior a la media del sistema y que limita al resto de componentes. 56

DELETE Petición HTTP que permite eliminar un recurso del servidor. Transmite los datos desde la url. 6

Dell Wyse Xenith Clientes thin propiedad de Dell. 9

dependencias Restricciones de inicio y finalización de una tarea con respecto a otra. Son: Comienzo Comienzo (*una tarea no puede comenzar hasta que la otra comience*), Comienzo Fin (*Una tarea no puede comenzar hasta que la otra finalice*), Fin Comienzo (Una tarea no puede finalizar hasta que la otra comience) y Fin Fin (Una tarea no puede finalizar hasta que la otra finalice). 19, 21

descriptores En el contexto actual, índice que identifica un fichero dentro de la tabla de ficheros abiertos por un proceso. 5, 18, 46, 66

device driver Un device driver (en español controlador de dispositivo) es un software que conoce como funciona un hardware específico y es capaz de interactuar con él. Suele actuar como interfaz entre el software del usuario y el dispositivo que maneja. En la mayoría de kernels los device drivers se integran dentro de su propio código en caliente. El desarrollo de device drivers es una de las labores de programación mas complejas. 2, 11, 18

diccionario En este contexto: Estructura de datos que consta de pares *clave valor*. 12

Docker Proyecto de código abierto para automatizar el despliegue de aplicaciones dentro de contenedores de software, aislados del sistema operativo en su propio espacio de usuario. 3, 5, 10, 11, 13, 27, 36, 37, 41, 44, 62

Dockerfile Fichero de configuración de imágenes empleado por Docker para componer los contenedores. 30, 44

dominio Identificador amigable de una red o máquina específica. 8, 12, 39

drawio Software de dibujo propiedad de Diagramsnet. 15

echo Comando Linux que muestra un texto especificado por pantalla. 50

elasticidad En este contexto: capacidad de incrementar y disminuir dinámicamente los recursos en base a la demanda. 3, 31

endpoint Dirección virtual que, al ser invocada, ejecuta una función. 6, 17, 43, 45, 46, 49, 50, 54, 55

enrutador Máquina capaz de recibir paquetes de red por una interfaz, establecer un camino hacia su destino y reenviarlos. 12

ensamblador Lenguaje de programación con equivalencia directa entre su código y las instrucciones máquina de una CPU. 31

escalabilidad Capacidad de un sistema de aumentar sus prestaciones a lo largo del tiempo. Existen dos tipos de escalabilidad: vertical (las prestaciones aumentan con la mejora del hardware actual) y horizontal (las prestaciones aumentan con la agregación de recursos). 3, 7, 28, 29, 64

espacio de nombres Mecanismo que permite aislar a un proceso del resto, haciéndole creer que cuenta con exclusividad sobre el equipo. Nos permite, por ejemplo, que un proceso vea a otro con el PID 0 sin ser este el scheduler del kernel. 6

ext4 Sistema de ficheros transaccional. Es ámpliamente utilizado con Linux. 62, 63

firewall Ver cortafuegos. 12, 29, 33, 36

firmware Programa encargado de manejar el hardware de un dispositivo para un propósito mas específico que el kernel. Normalmente el firmware no está diseñado para ser modificable. 2, 9, 65

Flag En informática, parámetro que altera el funcionamiento de un programa. 46

Flask Framework para desarrollo web. 6, 12, 43, 52, 63

forward only Directiva que indica que solo emplearemos el servidor DNS como caché. 35

frontend Elementos de una página web que se ejecutan en el navegador del cliente. 26, 62

Gantt Diagrama que muestra las diferentes tareas de un proyecto, sus relaciones y dependencias. 21

GET Petición HTTP que solicita un recurso y transmite datos mediante la propia url. 6, 46

GitHub Servicio de alojamiento para proyectos software propiedad de Microsoft. 33, 50

Google Stadia Servicio de streaming de videojuegos propiedad de Alphabet Inc. 9

Gunicorn Servidor de aplicaciones web con soporte para python. 12

HALTED El estado *halt* de una CPU es un estado de espera en el que el pipeline ”duerme” hasta recibir una interrupción. 65

hashes Resultado de aplicar una función matemática repetible que convierte una entrada de cualquier tamaño a un texto de longitud específica. 31

head Cabezera de un documento HTML. Contiene entre otros, el título y los estilos del sitio. 51

hilo de ejecución Código que se ejecuta de manera concurrente a otro dentro de un mismo programa. 5

horas-hombre Horas de esfuerzo asociadas a un recurso humano. 18

HyperV Software de virtualización de Microsoft. 5

IBM Serie Z Familia de computadoras mainframe desarrolladas por IBM que comienza con la z900. 7

imagen En este contexto es el resultado de compilar una Dockerfile, lista para generación de contenedores. 27, 30, 35, 37, 41

iniciador iSCSI En terminología iSCSI es quien solicita almacenamiento al target. 13, 26, 38

instancia En este contexto: máquina virtual Docker ejecutandose. 17, 18, 30, 41–45, 53–55, 57, 64

Intel SHA Extensions Conjunto de instrucciones incluidas en los procesadores Intel modernos dedicadas a la aceleración de las operaciones de hashing mediante Secure Hash Alogorithm. 31

interfaz de red Interfaz física o virtual situada en los extremos de una comunicación de red. Lleva asociada la dirección MAC. 12, 35

interrumpe Una interrupción hace referencia a la alteración del flujo de ejecución de un procesador debido a la llegada de un evento que debe ser atendido. 65

IPs dinámicas Dirección IP que es asignada por el ISP durante un periodo, que rota en de manera aleatoria. 8

IPTables Software de cortafuegos para sistemas Linux propiedad de Netfilter. 12

ISC dhcp server Software de servidor DHCP propiedad de ISCorg. 12, 34

iterativo incremental Metodología de desarrollo que aplica las fases del desarrollo en cascada a pequeñas subdivisiones del proyecto, ordenadas por importancia. 17

Jinja2 Motor de renderizado de plantillas web integrado en Flask. 12, 52

JQuery Librería para el lenguaje de programación JavaScript que facilita muchas de sus operaciones comunes. 11, 47, 51

JQuery-UI Librería de estilos que extiende algunas funcionalidades de JQuery. 11

kernel Programa encargado de manejar los recursos *hardware* del sistema y actuar como interfaz entre este y el usuario. 2, 18, 65, 66

latencia Tiempo que tardamos en recibir respuesta de un servidor desde que enviamos la consulta. 58

launcher Interfaz que permite acceder a las aplicaciones instaladas en un sistema. 65

lenguajes interpretados Lenguaje de programación que ejecuta su código mediante un programa externo, denominado intérprete. 20

linuxserver.io Organización dedicada a la elaboración de imágenes de distribuciones Linux. 10

lista blanca Lista de clientes a los que permitimos conectarse a un servidor. 30

localhost Nombre de dominio que hace referencia a la dirección de la interfaz de *loopback* de una máquina. 31

Login Form V16 Plantilla de incio de sesión libre propiedad de Colorlib. 12, 44

línea base Agrupación de los caminos críticos (tareas sin holgura) de un proyecto. 18, 24

magic quotes Proceso de filtrado para los datos de entrada de un programa php que trata de evitar la inyección de código. 29

mainframe Computador de altas prestaciones, normalmente con características como permitir sustitución de módulos en caliente, redundancia y suministro eléctrico ininterrumpido, empleado como nodo de cómputo principal de una organización. 5, 7, 8

memoria virtual Mecanismo que simula de cara al software, un rango de direcciones de memoria independiente del estado de la memoria física del sistema. Esto permite a cada programa trabajar de manera independiente a su posición, el cambio de contexto eficiente, el Copy On Write y la protección de memoria entre otras muchas ventajas. Incialmente se desarrollo para permitir a los programadores emplear el almacenamiento secundario para almacenar procesos en espera en vez de la escasa memoria principal. 5

mkfs Comando linux para la creación de sistemas de ficheros. 14, 63

monolítico En este contexto: arquitectura según la que se entiende al kernel como un único programa, ejecutándose en modo supervisor y siempre presente en memoria principal. Linus torvalds, siguiendo la filosofía expuesta por Maurice Bach en su libro *Design of the UNIX Operating System* implementó Linux como un kernel monolítico. Esta decisión de diseño no se vió exenta de polémica, figuras partidarias del *microkernel* como Andrew S. Tanenbaum. Como curiosidad aún se puede consultar un extracto del acalorado debate que mantuvieron en el año 1992.. 2, 31

mount Comando Linux para el montaje de discos. 14

MS Project Software de planificación de proyectos propiedad de Microsoft. 15, 19, 66

muestreo Selección de un conjunto de valores discretos desde una señal analógica. 2

Multiseating Uso de un equipo físico por parte de múltiples usuarios conectados físicamente. 6

MySQL Conjunto de herramientas para crear y administrar servidores SQL propiedad de Oracle. 11, 13, 30, 35

máquinas virtuales Este término suele hacer referencia a un equipo PC virtualizado, sobre el cual se instalará un sistema operativo. 3, 5, 11, 14, 20, 55

nameserver Campo de configuración de Linux que permite especificar el servidor DNS a emplear. 38

nopasswd Política de configuración para servidores SSH que deshabilita el inicio de sesión mediante contraseña. 30

notificaciones push Notificaciones enviadas por un servidor hacia el cliente. 65

open iscsi Software libre de administración de iniciadores iSCSI propiedad de Open-ISCSI. 13, 37

OpenAPI Estandar de definición de APIs propiedad de la inciativa OpenAPI. 15, 28

openssh server Software servidor SSH para sistemas Linux propiedad de OpenBSD. 13, 14

Osboxes.org Sitio web que distribuye imágenes preconfiguradas de sistemas operativos para su uso con máquinas virtuales. 14, 35, 37

Overleaf Software de edición de documentos LaTeXen línea propiedad de Overleaf. 15

parser Un parser o analizador sintáctico es un algoritmo capaz de procesar una cadena de símbolos en base a un conjunto de reglas de una gramática formal. 4, 17, 45, 52, 61

PATCH Petición HTTP que modifica parcialmente un recurso ya existente en el servidor. Transmite los datos en el cuerpo de la petición. 6

PermitRootLogin Campo de configuración del servidor SSH que indica si se puede iniciar sesión como superusuario. 38

POST Petición HTTP que permite modificar o crear un elemento en el servidor. Transmite los datos en el cuerpo de la petición. 6, 46

Principio de Responsabilidad Única Este principio afirma que un módulo software ha de tener una función única, acotada y autocontenida. Está incluido en la lista de principios *SOLID*. 46

proceso En este contexto: programa informático una vez ha sido cargado en memoria. 5, 18

pruebas de ruptura Prueba cuyo objetivo es determinar la capacidad máxima que soporta un sistema sobrecargándolo progresivamente hasta que falle. 55

PulseSecure VPN Software de tunneling propiedad de Pulse Secure LLC. 8

PUT Petición HTTP que crea un nuevo recurso en el servidor de manera idempotente. Transmite los datos en el cuerpo de la petición. 6

red interna En VirtualBox, permite crear una nueva red virtual dedicada solo a los dispositivos que introduzcan su identificador. 34, 35, 37

renderizar generar una imagen a partir de su descripción no visual. 1

requisitos funcionales Lista de acciones que debe ser capaz de llevar a cabo un programa. 27

requisitos no funcionales Cualidades que ha de poseer un software, definen ”como debe hacer las cosas”. 26, 28

RESTful API que sigue los principios de la arquitectura REST *interfaz única, sin estado, cacheable, etc.* 15, 66

root Usuario privilegiado. 35

rsync Herramienta para la copia de ficheros del sistema Linux. 55

Scapy Librería para la creación y manipulación de paquetes de red para Python propiedad de Philippe Biondi. 41

sentencias insert En SQL, sentencia que permite añadir una nueva fila dentro de una tabla de datos. 36

Shell Scripting Lenguaje interpretado que agrupa comandos de shell linux. 11

sockets Canal de comunicación entre dos procesos. 11, 18

SQL injection Técnicas de ataque que consisten en incluir sentencias SQL entre los datos enviados a un servidor. 29

SqliteDict Librería de interfaz con bases de datos sql de python. 12, 46

stack de red En este contexto: Conjunto de programas encargados de manejar las comunicaciones de red del equipo, normalmente cuenta con software encargado de interactuar con las capas 1 a 4 del modelo OSI. 2

stderr Ver 2. 18, 46, 54

stdin Ver 0. 46

stdout Ver 1. 18, 39, 46, 50, 54

streaming (Anglicismo) Retransmisión. 9

submit Botón que representa la acción de enviar un formulario. 12

Swagger Herramienta de modelado de APIs en línea propiedad de SmartBear Software. 15, 49

swap Fichero alojado en memoria secundaria en el que se vuelcan datos desde la RAM que no estén siendo empleados en la actualidad con el fin de liberar espacio. 55–58

tabla de páginas Estructura de datos empleada por el sistema operativo, el procesador y la unidad de manejo de memoria (MMU) para la traducción de direcciones virtuales en direcciones virtuales. 5

tarantella Solución webtop desarollada por SCO en el año *1993*. 10

target iSCSI En terminología iSCSI es la agrupación de unidades lógicas. 54

templates En este contexto frontend de cada aplicación que ejecuta Clobber. 6, 12, 17, 47, 52, 63

Teradici PCoIP Cliente thin propiedad de Teradici. 9

terminales Dispositivo físico o virtual, que permite interactuar con un sistema informático. 5, 7

TGT (Linux Target Framework) Conjunto de utilidades para la administración de targets SCSI en Linux. 11, 14, 37

thick-provisioned Discos de tamaño inmutable independientemente de su espacio libre. 62

Thin o Zero Equipo de bajas prestaciones que se contectará a un servidor externo que contiene el sistema operativo y los datos. 9

thin-provisioning Discos cuyo tamaño migra en función de la demanda. 62

Tolerancia a fallos Conjunto de técnicas cuyo objetivo es permitir a un sistema seguir operando en presencia de errores. 7, 8, 28, 64

tubería En este contexto: mecanismo de intercomunicación de procesos bajo el paradigma de paso de mensajes. 5, 66

tunneling Técnicas empleadas para establecer un canal de comunicación cifrada entre dos puntos sobre internet. 8

UNIX Time-Sharing Espacio de nombres que contiene entre otros el nombre del equipo y el dominio al que pertenece. 6

val Parámetro de algunos componentes HTML que contiene su estado actual. Por ejemplo el texto de un <input type=”text”> . 47

VirtualBox Software de virtualización propiedad de Oracle. 5, 14, 18

virtualización Técnicas que permiten simular un dispositivo hardware mediante código. 1, 3, 5, 7, 9, 10, 13, 17, 26, 28, 29, 31, 42, 43, 53, 58, 64

VMWare Software de virtualización propiedad de EMC Corporation. 5, 14

VMWare Horizon Servicio de escritorio remoto a través del navegador propiedad de EMC Corporation. 9

volúmenes unidad de almacenamiento identificada mediante una etiqueta. 41

webtop Entorno de trabajo virtual accesible desde un navegador. 10

with Bloque de código que abre y cierra un fichero de recursos de manera automática. 11

x86 Conjunto de instrucciones para microprocesadores diseñada por Intel. 31

Xhepyr Software de Multiseating para Linux propiedad de la X.Org Foundation. 7

zip Formato de compresión sin pérdida de ficheros. 12, 45

Zipfile36 Librería para el manejo de ficheros comprimidos de python. 12

Bibliografía

[1] C. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, jan 1949. [En línea]. Disponible en: https://doi.org/10.1109/jrproc.1949.232969

[2] R. J. Pumphrey, "Upper limit of frequency for human hearing," *Nature*, vol. 167, no. 4246, p. 438–439, 1951.

[3] L. Rice, "Containers from scratch," 2019, consultado el 20 de abril de 2021. [En línea]. Disponible en: https://gotoams.nl/2018/sessions/429/containers-from-scratch

[4] H. Mason, "Scsi, the industry workhorse, is still working hard," *Computer*, vol. 33, no. 12, pp. 152–153, 2000.

[5] N. Bontis and H. Chung, "The evolution of software pricing: from box licenses to application service provider models," *Internet Research*, 2000.

[6] N. Zlatanov, "The data center evolution from mainframe to cloud," *IEEE Computer Society*, 2016.

[7] A. SIlkalns, "Login form v16," 2021, consultado el 5 de junio de 2021. [En línea]. Disponible en: https://colorlib.com/wp/template/login-form-v16/

[8] hua, "iptable port forward between two lan interface," 2012, consultado el 20 de junio de 2021. [En línea]. Disponible en: https://www.linuxquestions.org/questions/linux-server-73/iptable-port-forward-between-two-lan-interface-945719/

[9] Gewure, "Map a range of ports to another range of ports (equal lengths of ranges)," 2019, consultado el 21 de junio de 2021. [En línea]. Disponible en: https://askubuntu.com/questions/1108042/map-a-range-of-ports-to-another-range-of-ports-equal-lengths-of-ranges

[10] maff1989, "Dnat port range with different internal port range with iptables," 2016, consultado el 21 de junio de 2021. [En línea]. Disponible en: https://serverfault.com/questions/729810/dnat-port-range-with-different-internal-port-range-with-iptables

[11] E. Glass, "Cómo instalar el servidor web apache en ubuntu 20.04," 2020, consultado el 28 de mayo de 2021. [En línea]. Disponible en: https://www.digitalocean.com/community/tutorials/how-to-install-the-apache-web-server-on-ubuntu-20-04-es

[12] V. Gite, "Linux tgtadm: Seetup iscsi target (san)," 2008, consultado el 20 de mayo de 2021. [En línea]. Disponible en: https://www.cyberciti.biz/tips/howto-setup-linux-iscsi-target-sanwith-tgt.html

[13] E. Ivanec, "How to make iptables rules expire?" 2011, consultado el 20 de junio de 2021. [En línea]. Disponible en: https://serverfault.com/questions/273324/how-to-make-iptables-rules-expire

[14] M. Costa, "How to get a docker container ip address - explained with examples," 2020, consultado el 5 de junio de 2021. [En línea]. Disponible en: https://www.freecodecamp.org/news/how-to-get-a-docker-container-ip-address-explained-with-examples/

[15] BMitch, "How to check if the docker engine and a docker container are running?" 2017, consultado el 5 de junio de 2021. [En línea]. Disponible en: https://stackoverflow.com/questions/43721513/how-to-check-if-the-docker-engine-and-a-docker-container-are-running

[16] code_monk, "Get docker container id from container name," 2015, consultado el 5 de junio de 2021. [En línea]. Disponible en: https://stackoverflow.com/questions/34496882/get-docker-container-id-from-container-name

[17] Y. Singh, "Create simple login page with php and mysql," 2020, consultado el 6 de junio de 2021. [En línea]. Disponible en: https://makitweb.com/create-simple-login-page-with-php-and-mysql/